Bruno Miguel de Figueiredo Carvalho

Licenciado em Engenharia Informática

Monitorização de Defeitos em Runtime

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Rodrigo Serafim, CTO, Quidgest

Co-orientadora: Armanda Rodrigues, Profª. Auxiliar, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Janeiro, 2020

Monitorização de Defeitos em Runtime

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Resumo

Sistemas de grande dimensão não são facilmente desenvolvidos ou mantidos. Em certos casos é difícil encontrar problemas no código, realizando apenas uma análise estática. A maneira mais comum de lidar com este problema é através do uso de debuggers. No entanto, pode não ser fácil, por exemplo, testar todos os valores de input possíveis ou testar código que interaja com subsistemas complexos ou com código desconhecido.

De forma a encontrar problemas na execução de sistemas complexos, é necessário realizar uma monitorização contínua. Grande parte dos sistemas implementa registos de eventos. Estes registos contêm informação acerca do estado de cada subsistema, que podem ser usados para inferir o estado geral do sistema.

Nesta dissertação propõe-se uma solução de monitorização modular, flexível e configurável, baseada em registos de eventos de diferentes sistemas. Embora esta solução tenha sido desenvolvida no âmbito de um projeto específico com necessidades específicas, pretende-se que esta possa ser adaptada para outros casos. De modo a facilitar o desenvolvimento futuro, ao longo deste documento, irá ser descrita toda a implementação atual, a sua estrutura e ainda, a análise realizada para a escolha de ferramentas e para outras tomadas de decisão.

De modo a avaliar a solução implementada, foram realizados alguns testes, comparando o seu desempenho com uma outra solução semelhante, implementada anteriormente, que se pretendia melhorar. Para avaliar a sua utilidade, foi ainda pedido a algumas pessoas que testassem a nova solução e a comparassem com a solução antiga.

Palavras-chave: Monitorização, *Debugging*, *Logging*, *Health Check*

Abstract

Systems with big dimensions are not easily developed or maintained. In some cases, it is hard to find every problem in the code by only executing a static analysis. The most common way to deal with this problem is by using debuggers, however, these tools also have some limitations, for example, it might not be possible to test every possible combination of input values, or to test code that interacts with complex systems or with unknown code.

In order to find problems in the execution of complex systems, it is necessary to perform a continuous system monitoring. Most systems implement some type of logging. Log files contain useful information about the state of each subsystem, this information can then be used to infer the state of the main system.

This dissertation proposes a flexible and configurable solution for monitoring, based on logs from different systems. This solution was developed in the context of a specific project with specific needs, however, it should be possible to be adapted to be used in other projects or systems. In order to help further development, in this document it will be described the development, structure and even the research and analysis executed in order to choose different tools or ways to implement the final solution.

To evaluate this solution, some tests were conducted in order to compare its performance against another similar solution that was previously developed. To evaluate its usefulness, some people were asked to test the final solution and to compare it with the previous one.

Keywords: Monitoring, Debugging, Logging, Health Check

Índice

Lista de Figuras

Listagens

Capítulo 1

Introdução

Neste capítulo será apresentado o projeto desta dissertação, a motivação por detrás do mesmo, o problema que este pretende resolver e de que forma irá ser resolvido. Ainda neste capítulo será apresentada a estrutura deste documento.

1.1 Motivação

O desenvolvimento tecnológico verificado nas últimas décadas possibilitou a criação de vários tipos de dispositivos. Especialmente devido ao crescimento dos dispositivos móveis, existe uma enorme facilidade em aceder a diferentes sistemas, desde os mais comuns, como jogos e redes sociais, até mesmo ferramentas de trabalho. Este fator, aliado a uma necessidade atual, de várias empresas e negócios, de se destacarem num mercado cada vez mais competitivo, resulta no rápido desenvolvimento de grandes sistemas com um grande número de funcionalidades.

Os sistemas informáticos atuais são desenvolvidos através de vários milhares de linhas de código. Desta forma, torna-se bastante difícil manter, corrigir ou melhorar todas a partes de um sistema de grande dimensão. De forma a facilitar estas ações de manutenção, um sistema pode ser dividido em subsistemas mais pequenos que se focam na implementação e manutenção de apenas um subconjunto das funcionalidades do sistema final. Esta divisão permite a criação de equipas ou empresas especializadas na manutenção desses subsistemas. Adicionalmente, por norma, é criada uma interface para a ligação com o sistema principal, criando assim uma camada de abstração para os detalhes da sua implementação.

Um sistema, mesmo com várias funcionalidades, pode não ser especialmente útil, caso existam erros ou problemas de otimização, que ponham em causa a disponibilidade do serviço. Encontrar erros ou pontos a otimizar durante o desenvolvimento de sistemas pode

não ser uma tarefa fácil, principalmente quando se trabalha com subsistemas cujo código é desconhecido ou demasiado complexo. Nem sempre é percetível que certas ações podem ter elevada complexidade ou que, em casos específicos, possam mesmo causar erros.

Vários erros ou atrasos podem ocorrer durante a execução de um sistema. De modo a detetá-los, será necessário realizar uma monitorização do sistema de forma contínua. Uma das formas mais comuns de efetuar este tipo de monitorização é através da utilização de *debuggers* e da execução de alguns testes ao sistema. No entanto, ainda que estas técnicas sejam úteis para um pequeno conjunto de funcionalidades, testar todas as combinações de pedidos e *input* não é possível. Além disso, estes processos de *debugging* são muito pesados em termos de desempenho.

Um outro tipo de monitorização pode ser conseguida através de análise de registos de eventos. Periodicamente ou sempre que aconteça algo específico, como por exemplo um acesso de um cliente, a execução de uma chamada à base de dados, a ocorrência de um erro, etc., é possível registar informação acerca desses eventos para ser analisada posteriormente. Algumas vantagens deste tipo de monitorização são que, uma vez armazenados os registos, estes podem ser processados em paralelo à execução do sistema, tal como já foi estudado em [1], e ainda, por serem menos intrusivas relativamente à execução do sistema, estas soluções podem ser usadas durante todo o tempo em que o sistema está a executar sem comprometer o seu desempenho.

1.2 Descrição do Problema

Na secção anterior foi apresentada uma necessidade de realizar uma manutenção durante a execução do sistema. Foram também apresentadas as vantagens de soluções de monitorização, e especialmente, soluções baseadas em registos de eventos. No entanto, estas soluções podem trazer alguns problemas adicionais.

Primeiramente, os sistemas de registos de eventos têm potencial para gerar uma grande quantidade de registos num espaço de tempo relativamente curto. Deste modo, é necessário processar esses registos, analisar a relevância da informação neles contida, de modo a apenas guardar a que for estritamente necessária, e ainda, analisar soluções de armazenamento, de modo a que o armazenamento, a leitura e o processamento desses dados seja o mais eficiente possível.

De modo a obter informação acerca do estado do sistema geral, é necessário monitorizar os subsistemas e, de alguma forma, conseguir relacionar toda essa informação de modo a poder tirar conclusões. O grande problema é que subsistemas diferentes são possivelmente independentes, e, desse modo, cada um deles pode gerar registos com formatos e informação diferentes entre si. Para solucionar este problema, deverá ser efetuada uma análise aos dados incluídos nos registos das diferentes fontes, de modo a tentar estabelecer ligações entre os eventos dos vários sistemas.

Embora informativa, toda esta informação poderá perder utilidade se não existir uma

forma eficiente de a analisar. Para tal, deverá ser disponibilizado um ambiente de monitorização, onde se possa consultar a informação proveniente dos registos de eventos. Seria interessante que, para além de informação sobre o estado do sistema num momento no tempo, pudesse ser consultada informação estatística, como por exemplo, relativamente aos pedidos de cliente recebidos num servidor web, o tempo médio da sua execução, número médio de pedidos num determinado espaço de tempo, etc..

1.3 Contexto Prático

De modo a analisar estas questões de uma forma mais concreta, o trabalho a realizar para esta dissertação foi desenvolvido num ambiente empresarial e num contexto específico na Quidgest. A Quidgest[1] é uma empresa que fornece serviços de consultoria e desenvolve soluções para diversos tipos de clientes. O sistema estudado neste projeto de dissertação é o sistema integrado de gestão para a Universidade NOVA de Lisboa, que se encontra disponível, para pessoal autorizado, através de um portal web.

O sistema estudado é constituído por vários subsistemas. Sendo este um sistema para uma aplicação web, inclui um sistema responsável pela comunicação HTTP, um outro sistema responsável pelo processamento de dados e um outro sistema responsável pelo armazenamento de informação. Todos estes subsistemas podem ser configurados para gerar registos de eventos que posteriormente serão recolhidos e processados num ambiente de monitorização independente.

Previamente ao início deste projeto, a Quidgest implementou uma solução provisória para a monitorização do sistema. Esta solução inclui um painel de controlo onde é apresentada informação estatística acerca do historial de eventos. Periodicamente, o sistema de monitorização, abre uma conexão com o servidor e descarrega todos os novos registos de eventos e guarda-os numa base de dados relacional. Estes dados são usados por um software de apresentação de um painel de controlo, para construir e apresentar vários gráficos e tabelas. Esta solução, no entanto, tem alguns problemas que dificultam a sua utilização. Um dos principais problemas é que, visto que a informação contida nos registos de eventos, não é processada previamente, esta, por um lado, gera um grande peso no software de apresentação do painel de controlo, causando erros de *timeout*, por outro lado, a própria informação não está a ser corretamente processada a partir dos ficheiros de registos de eventos, o que por vezes resulta na inserção de informação inválida em certos campos do registo na base de dados.

1.4 Objetivos

O objetivo principal deste projeto de dissertação é desenvolver uma solução de monitorização contínua genérica, que permita melhorar a solução de monitorização provisória

[1] https://www.quidgest.pt/

implementada pela Quidgest. Este objetivo pode ser dividido em subobjetivos que serão seguidamente descritos:

- Analisar ficheiros de registo de eventos provenientes de diferentes sistemas, com o objetivo de definir uma estrutura de dados para cada tipo de registo.
- Analisar diferentes tipos de soluções de armazenamento no contexto deste problema, com o objetivo de escolher, instalar e configurar uma dessas soluções de forma a melhorar o desempenho da solução geral.
- Implementar uma solução de processamento automático de ficheiros de registos de eventos provenientes de diferentes sistemas. Esta solução deverá:
  - Estruturar a informação dos ficheiros no formato definido anteriormente;
  - Armazenar a informação processada no sistema de armazenamento escolhido;
  - Ser implementada de modo a facilitar desenvolvimento e configuração futuros.
- Implementar e/ou configurar uma ferramenta de visualização de dados que permita ler e analisar a informação dos registos com uma maior facilidade. Esta ferramenta deverá permitir:
  - Criar um painel de controlo conjugando vários tipos de visualização de dados;
  - Definir filtros sobre os dados a serem apresentados.

Terminado o período de desenvolvimento deste projeto de mestrado, pretende-se que seja possível fornecer à Quidgest uma ferramenta funcional que permita melhorar o processo de detetar falhas e as suas causas. Embora o desenvolvimento desta solução seja aplicada a um projeto específico, pretende-se que a sua implementação, bem como toda a informação presente neste documento, facilitem, tanto alterações posteriores da solução desenvolvida, bem como, permitir a possibilidade de ser adaptada para outros projetos com diferentes sistemas e configurações.

1.5 Organização do Documento

Nesta secção irá ser apresentada a estrutura e organização do presente documento. Este documento está organizado em capítulos, secções e subsecções, sendo que, em alguns casos, existem níveis adicionais de subsecções dentro de outras subsecções.

O primeiro capítulo serve de introdução ao projeto de mestrado desenvolvido. Na primeira secção são apresentados os principais motivos que deram origem a este projeto, estes motivos são explicados numa ordem específica de modo a estabelecer uma ligação para a segunda secção. Na segunda secção é apresentado o problema que se pretende resolver de uma forma geral. A terceira secção apresenta o contexto prático onde este projeto foi desenvolvido. A quarta secção serve como uma introdução mais específica

ao projeto e são indicados os seus objetivos. A quinta secção apresenta a estrutura e organização do documento.

O segundo capítulo apresenta uma análise, não só das tecnologias e ferramentas utilizadas no desenvolvimento, mas também do trabalho já realizado anteriormente, por diversos autores, que esteja de algum modo relacionado com o trabalho a desenvolver neste projeto.

Na primeira secção são apresentados trabalhos anteriormente realizados e relacionados com o trabalho a desenvolver nesta dissertação, entre eles, a solução de monitorização provisória implementada pela Quidgest. Cada um dos trabalhos será descrito na sua respetiva subsecção. A segunda secção apresenta o estudo realizado de modo a comparar diferentes sistemas de armazenamento de informação. São apresentadas as condições em que os diversos testes foram executados, bem como que algoritmos foram usados para cada sistema. No fim da secção será indicado o sistema escolhido e as razões para essa escolha. A terceira secção apresenta uma comparação entre algumas soluções de apresentação, e as razões pela escolha daquela que foi usada na implementação. Na quarta secção são apresentadas as tecnologias usadas em mais detalhe. Por fim, a última secção deste capítulo apresenta um resumo do conteúdo tratado no mesmo.

O terceiro capítulo descreve o desenvolvimento da solução proposta. Sendo que este desenvolvimento foi realizado em várias fazes, cada uma destas fases será descrita na sua respetiva secção.

A primeira secção descreve a estrutura da solução, tanto a estrutura global, como também a estrutura de todo o código desenvolvido. Neste capítulo também é feita uma comparação de estruturas entre a solução provisória e a nova solução implementada durante o desenvolvimento deste projeto, sendo indicadas as principais diferenças entre as duas. O objetivo principal desta secção é orientar o leitor durante o resto do documento onde será explicado o processo de desenvolvimento baseado na estrutura apresentada.

A segunda secção descreve a análise realizada a diversos tipos de ficheiros de registos. A partir dos resultados desta análise é ainda definida a estrutura de dados final para os tipos de registos analisados, bem como o processo de configuração do processamento de modo a reestruturar a informação original para que possa ser armazenada usando a estrutura definida.

A terceira secção descreve de que forma foi implementado o processamento dos registos de eventos. Desde regras específicas de cada tipo analisado até regras gerais de processamento de ficheiros, será descrita toda a lógica bem como o código implementado.

A quarta secção descreve a implementação de vários elementos visuais da solução de apresentação, bem como de que forma é foi implementado o painel de controlo para a visualização da informação extraída dos registos.

Finalmente no último capítulo serão apresentados os resultados do projeto, algumas conclusões e sugestões para desenvolvimento de trabalho futuro.

Capítulo 2

Suporte Tecnológico e Trabalho Relacionado

Neste capítulo irão ser apresentadas as tecnologias usadas na solução provisória implementada pela Quidgest, bem como as que foram usadas durante o desenvolvimento deste projeto de mestrado. Será também apresentado algum trabalho relacionado com este projeto que foi realizado anteriormente por outros autores.

2.1 Trabalho Relacionado

Durante um período de preparação para a elaboração e desenvolvimento desta dissertação foram analisados alguns trabalhos relacionados e considerados relevantes no âmbito deste projeto. Nesta secção irão ser apresentados alguns desses trabalhos tal como a forma como os mesmos influenciaram a solução a desenvolver.

2.1.1 Monitorização Baseada em Registos de Eventos

No passado já foram realizados e estudados métodos de monitorização baseados na análise de registos de eventos. Em [1] é descrita uma solução de monitorização de um programa que corre em apenas um núcleo de processamento. Quando este programa gera um registo, este é comprimido por um hardware específico e enviado para uma fila em memória. Um segundo núcleo, periodicamente procura itens adicionados à mesma fila, retira-os da fila assim que possível, descomprime o conteúdo e processa a informação neles contida.

Esta estrutura é bastante semelhante à usada no sistema provisório de monitorização da Quidgest, e que é de igual forma semelhante à a solução final deste projeto, uma vez que permite o processamento da aplicação web e da solução de monitorização em paralelo.

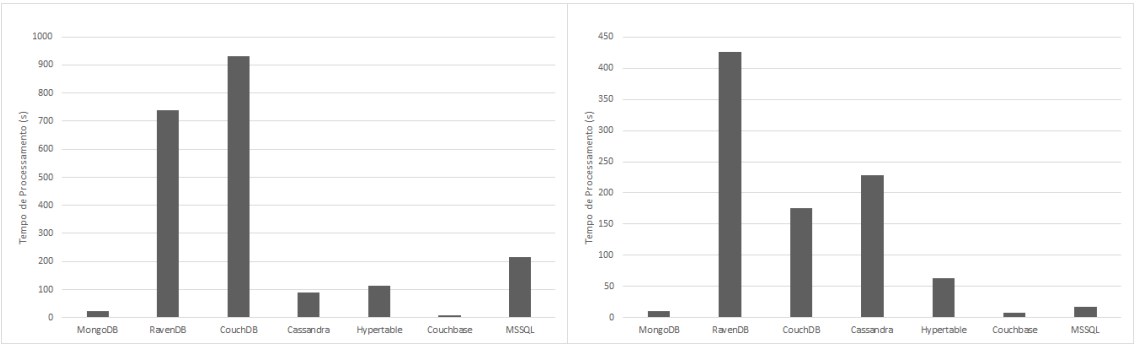
A maior diferença é que a solução para este projeto de mestrado é implementada em mais alto nível. Em vez de ser aplicada a um programa, esta solução será usada em subsistemas que compõem a aplicação web e, em vez de núcleos, a solução irá ser processada em máquinas diferentes.

Este trabalho foi posteriormente complementado em [2], considerando também o problema de desenvolvimento de programas que usam vários processadores. Neste tipo de programas, a necessidade de monitorização contínua, é ainda mais importante, visto que existe uma grande dificuldade em replicar a ordem das operações que causam erros. Embora existam diferenças entre este trabalho e o primeiro, a ideia principal é semelhante.

2.1.2 Comparação Entre Bases de Dados SQL e NoSQL

Existe algum trabalho anteriormente realizado onde são comparados, em termos de performance, vários sistemas de gestão de bases de dados, sendo esses do tipo SQL ou NoSQL, de modo a tentar perceber em que situações um é relativamente melhor que o outro.

Em [7] concluiu-se que, por norma, as bases de dados relacionais são relativamente mais lentas a executar leituras e escritas numa única tabela. A figura 2.1 representa graficamente os resultados obtidos pelos autores. Os gráficos presentes nesta figura indicam o tempo de processamento total relativo a 100000 operações para cada um dos tipos de operações e sistemas estudados.



a Teste de operações de escrita

b Teste de operações de leitura

Figura 2.1: Comparação de desempenho entre diferentes bases de dados[7].

Ao analisar os resultados obtidos no estudo apresentado anteriormente, conclui-se que existem vantagens em em usar soluções NoSQL para casos em que serão executadas várias operações de escrita e leitura em elementos de uma só coleção ou tabela. Nomeadamente, os sistemas MongoDB e Couchbase, apresentam os melhores resultados de desempenho entre os estudados.

2.1.3 Monitorização de Sistemas Usando Aprendizagem Automática

O artigo [6] é um artigo publicado pela NASA em 2004, onde é estudada uma solução de monitorização de sistemas baseada em registos de eventos e usando métodos de aprendizagem automática para deteção de anomalias.

O trabalho deste artigo foi realizado no âmbito do estudo do acidente aeroespacial que envolveu a nave STS-107 Columbia, que se desintegrou durante a reentrada na atmosfera, devido a um problema causado pela rutura da proteção térmica durante a subida da nave, causando a morte aos sete tripulantes. Durante 15 dias, a nave esteve em orbita com a sua proteção térmica comprometida sem o problema ser detetado. Com o sistema de monitorização apresentado neste artigo, este problema seria detetado quase imediatamente. Desta forma, a nave poderia ter sido reparada em órbita e possivelmente poder-se-iam ter poupado o material e, principalmente, as vidas dos tripulantes.

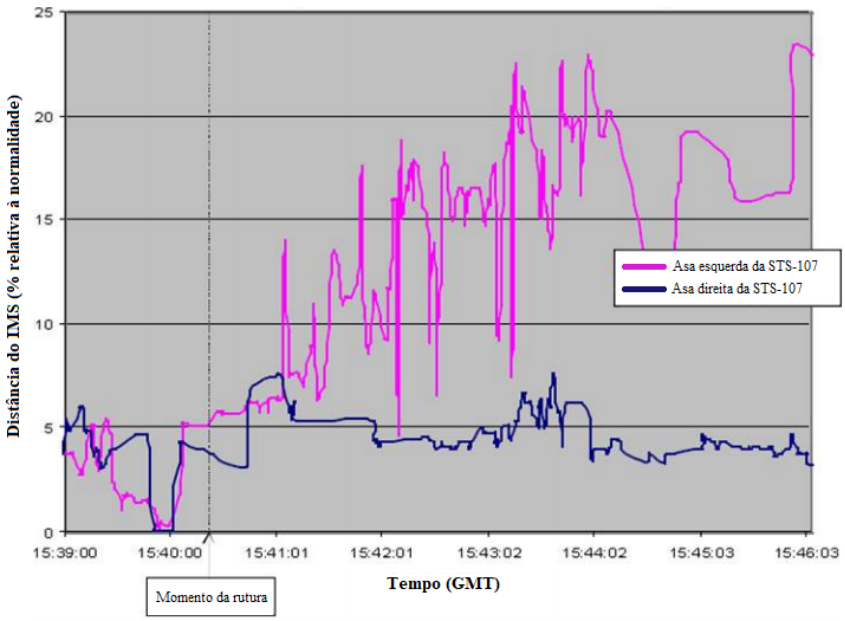


Figura 2.2: Resultados do sistema de monitorização (percentagem relativa à normalidade) relativamente ao tempo da medição. Gráfico adaptado de [6].

Na figura 2.2 são apresentados os resultados do sistema de monitorização para os valores dos sensores registados durante a subida da nave. Este gráfico relaciona o tempo em que foram lidos e a sua distância relativa à normalidade. Durante os primeiros momentos registados, os valores para ambas as asas da nave encontram-se bastante semelhantes. No entanto, após o momento da rotura da proteção térmica na asa esquerda, o valor relativo a essa asa começa a subir gradualmente, afastando-se consideravelmente da normalidade.

Este trabalho tem importância no âmbito do projeto a desenvolver durante esta dissertação, visto que demonstra um exemplo útil de uma solução que permite identificar situações de anormalidade, baseada em registos de eventos. Adicionalmente, ainda que não sejam estudadas soluções de aprendizagem automática, este projeto foi desenvolvido com este tipo de tecnologias em mente, desse modo, será relativamente simples

de adicionar futuramente, algoritmos de aprendizagem automática usando funções de pré-processamento de registos de eventos.

2.1.4 Monitorização na Quidgest

Como apresentado no capítulo de introdução, a Quidgest desenvolveu uma solução de monitorização provisória que possui alguns problemas. Nesta secção será apresentada essa solução em mais detalhe.

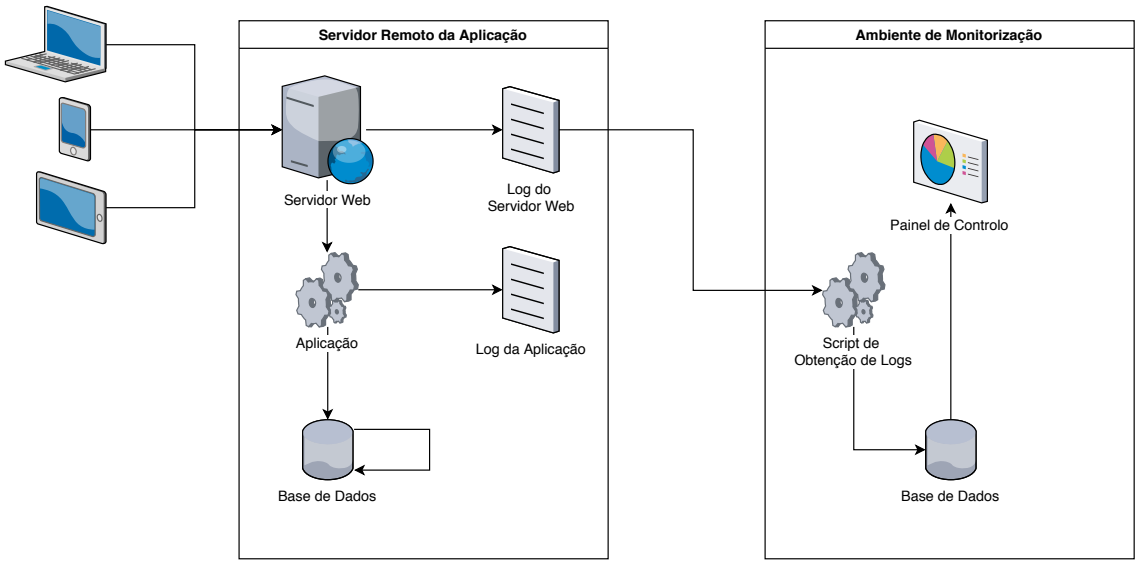


Figura 2.3: Estrutura da solução provisória de monitorização

A figura 2.3 representa a arquitetura provisória implementada na Quidgest, com o sistema da aplicação web, o sistema de monitorização e como estes se relacionam entre si. Os dois sistemas correm em máquinas diferentes. Este fator tem grande importância visto que, desta forma, é possível eliminar, quase por completo, qualquer impacto que a monitorização possa ter no sistema da aplicação (o único fator que pode influenciar o desempenho é o processo de obtenção dos registos, no entanto, visto que este processo corre periodicamente, a quantidade de informação a processar de cada vez, é relativamente reduzida).

Esta solução, utiliza um *script* PowerShell que analisa o ficheiro de registos de eventos do servidor IIS, seleciona os registos que ainda não foram processados, estabelece a comunicação com a base de dados no ambiente de monitorização, processa os registos, separando os diferentes campos, e guarda esses registos, de uma forma estruturada, na base de dados do sistema Microsoft SQL Server. Este *script*, no entanto, possui algumas limitações: a principal é o facto dos registos guardados serem apenas os registos do servidor IIS. Ainda que estes sejam os mais informativos relativamente a verificar existência de problemas, não são especialmente informativos sobre as causas desses problemas. Outro possível problema é que, bases de dados relacionais, podem não ser a melhor opção para guardar estes registos.

Após a obtenção e o armazenamento dos registos de eventos, é utilizado um software de apresentação para configurar um painel de controlo, onde são criados e dispostos vários gráficos, tabelas e outras formas de apresentação de dados. A métrica mais analisada neste painel de controlo é o tempo de execução de pedidos. Esta métrica é analisada através de gráficos de linhas que mostram a evolução do tempo médio de pedidos ao longo do tempo. Tal como referido na introdução, este sistema, por vezes, não funciona corretamente devido à quantidade de informação que é processada.

2.2 Análise de Soluções de Armazenamento

Nesta secção irá ser apresentado todo o trabalho realizado sobre a análise de possíveis soluções de armazenamento para armazenar os registos de eventos recolhidos do sistema a monitorizar. Estas soluções são responsáveis, tanto pelo armazenamento dos dados, como também por realizar algumas operações de agregação, úteis para quando for implementada a visualização dos dados. Os sistemas considerados no âmbito deste projeto usam técnicas diferentes para guardar e ler dados, o mais comum é o sistema MySQL que implementa uma base de dados relacional, no entanto, também irá ser analisada um sistema No-SQL, o MongoDB que promete melhores velocidades de escrita e leitura, e ainda, um sistema relativamente novo, o Elasticsearch, um motor de busca que promete um bom desempenho em operações de agregação de dados. Após a análise destes sistemas, foi escolhido aquele que melhor se adequa às necessidades deste projeto.

2.2.1 Características do Ambiente de Testes

Todos estes sistemas foram testados no mesmo ambiente e nas mesmas condições. Para correr o código desenvolvido foi usada a versão 3.7.4 do Python para Windows. As principais características da máquina onde foi realizado todo o desenvolvimento e testes são as seguintes:

- **Sistema operativo**: Windows 10, versão Enterprise de 64 bits
- **Processador**: AMD FX-8120 com 8 núcleos e 3.10 GHz de frequência base
- **Memória**: 10GB DDR3 com uma frequência de 667 Mhz
- **Disco**: 240GB de armazenamento SSD com ligação SATA-III

Sobre esta máquina foi ainda configurada uma máquina virtual Linux, onde foi simulado o ambiente de monitorização e instalados todo o software necessário para o processamento de registos, bem como, os sistemas de armazenamento usados nos testes apresentados neste capítulo. As características do ambiente de monitorização são as seguintes:

- **Software de virtualização**: VMWare Workstation Player versão 15.1.0

- **Sistema operativo**: CentOS Linux versão 7
- **Processador**: 2 núcleos de processamento alocados de 8 totais
- **Memória**: 4GB de memória alocada de 10GB totais
- **Disco**: 25GB de armazenamento alocado de 240GB totais

De modo a perceber que tipo de operações seria interessante analisar no estudo realizado, foi necessário perceber que tipo de operações são usadas pelo software de painéis de controlo que se pretende usar para apresentar visualmente a informação processada dos registos analisados. Um dos grandes problemas em apresentar este tipo de informação é que, devido ao elevado número de registos processados, existe muita informação para apresentar graficamente ao utilizador. Como forma de resolver este problema, muitas vezes, a informação ser apresentada no painel de controlo, é o resultado de operações de agregação de dados, assim, consegue-se apresentar muito menos informação sendo esta versão agregada dos dados igualmente informativa.

Assim, para além das operações básicas de escrita e leitura, foram também testadas operações de agregação de dados. De modo a fazer os testes consistentes para os diferentes sistemas, foram definidas regras para estas operações de agregação de dados. A operação a testar terá de agrupar todos os registos guardados, agrupá-los em grupos de 10 minutos e, para cada um desses grupos, calcular a soma dos tempos de processamento registados em todos os registos incluídos nesse grupo.

2.2.2 MySQL

MySQL é um dos sistemas de gestão de bases de dados mais usados atualmente[1]. Este sistema é usado para gerir bases de dados relacionais, através de comandos na linguagem SQL. Este tipo de sistemas são relativamente úteis quando se pretende relacionar vários tipos de informação, normalmente armazenada em diferentes tabelas. No caso estudado neste projeto, ainda que se pretenda relacionar informação de várias fontes, as principais ações a realizar na base de dados irão ser baseadas em operações de agregação de dados numa única tabela.

Embora alguns estudos, como o caso do artigo "A performance comparison of SQL and NoSQL databases"[7] analisado durante este projeto de mestrado, indicam que, normalmente bases de dados do tipo NoSQL têm melhor desempenho que as do tipo SQL. Ainda assim, este tipo de bases de dados foi considerado durante a análise de possíveis soluções de armazenamento a adotar para este projeto.

De modo a executar qualquer tipo de testes ou inserção de informação no sistema MySQL, é necessário definir previamente a estrutura a usar para armazenar os dados, essa estrutura irá ser igual à que foi estudada na secção 3.2. De modo a definir esta estrutura no sistema MySQL terá de ser usada a linguagem SQL para definir os nomes e

[1] https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/

tipos de campos numa query que será posteriormente processada pelo sistema. O código a executar será apresentado de seguida.

Listagem 2.1: Teste MySQL - Criar tabela

```
CREATE TABLE iis_log_test_db (
    id int(11) AUTO_INCREMENT,
    timestamp int(11) NOT NULL,
    client_ip varchar(15) NOT NULL,
    client_name varchar(45) DEFAULT NULL,
    server_name varchar(45) DEFAULT NULL,
    status varchar(3) NOT NULL,
    time_taken int(11) NOT NULL,
    sent int(11) NOT NULL,
    received int(11) NOT NULL,
    method varchar(10) NOT NULL,
    url_type varchar(45) NOT NULL,
    url_system varchar(45) DEFAULT NULL,
    url_language varchar(5) DEFAULT NULL,
    url_year varchar(4) DEFAULT NULL,
    url_form varchar(45) DEFAULT NULL,
    url_action varchar(45) NOT NULL,
    url_target varchar(255) NOT NULL,
    PRIMARY KEY(`id`)
) Engine=MyISAM
```

Neste código de definição de tabela existem alguns pontos importantes de serem analisados. Em primeiro lugar, o campo *timestamp* é armazenado como um tipo inteiro em vez de um tipo mais comum para este tipo de valores, como por exemplo, *datetime* ou mesmo *timestamp*, visto que o valor do tipo inteiro permite mais facilmente executar as operações aritméticas necessárias para realizar o agrupamento de registos. Outro ponto importante é o tipo de motor de armazenamento usado. Segundo a documentação oficial do sistema MySQL, o motor MyISAM tem um melhor desempenho do que o motor por omissão InnoDB, quando as operações executadas são maioritariamente leituras sobre uma só tabela[8]. Uma vez que neste caso, a maioria das operações que se pretendem executar são também de leitura sobre uma tabela, foi escolhido o motor MyISAM.

Inserção

O primeiro teste realizado foi o da inserção dos dados. Deste modo, é possível usar os dados inseridos para executar os restantes testes. Para este teste foi implementada uma função em Python para executar as inserções dos registos processados na tabela definida no sistema MySQL.

Listagem 2.2: Teste MySQL - Função de inserção

```
def insert(self, data):
    log_array = []
    for row in data:
```

```
4           log = [row[field] for field in self.structure]
5           log_array.append(log)
6
7       fields_str = ",".join(self.structure)
8       values_str = ",".join(["%s" for _ in range(0, len(self.structure))])
9       query = "INSERT␣INTO␣{}␣({})␣VALUES␣({})".format(
10          self.table_name, fields_str, values_str
11      )
12      self.cur.executemany(query, log_array)
13      self.db.commit()
```

Esta função cria e processa todos os valores necessários para construir a query que é depois enviada para o MySQL juntamente com a informação dos registos a inserir. Em primeiro lugar, esta função começa por criar uma lista de registos a lista *log_array*. Usando a variável *structure* guardada na classe, por cada registo, são iterados todos os campos existentes, os seus respetivos valores no registo são adicionados à lista *log* sendo esta lista depois adicionada à lista *log_array*. A variável *fields_str* guarda uma string com todos os campos da estrutura separados por vírgula. A variável *values_str* guarda uma string para definir a estrutura de cada registo da lista criada anteriormente. Esta string contém a string "%s"repetida, separada por vírgula, por cada campo presente nos registos. Estas duas ultimas variáveis, juntamente com a variável de classe *table_name*, são usadas para construir a query que será guardada na variável *query*. Por fim, usando um objeto de cursor de uma biblioteca de MySQL para Python, é executada a função *executemany* com os argumentos necessários para inicializar a inserção dos dados. Após a inserção dos dados é executada a função *commit* para confirmar as alterações realizadas.

Leitura

Para os testes de leitura é usada a função *read_all* bastante simples que apenas executa uma query básica para devolver todos os campos de todos os elementos da tabela.

Listagem 2.3: Teste MySQL - Função de leitura

```
1 def read_all(self):
2     self.cur.execute("SELECT␣*␣FROM␣{}".format(self.table_name))
3     return self.cur.fetchall()
```

Agregação

Para os testes de agregação é usada a função *group_by_10m*. Esta função é bastante semelhante à função anterior, no entanto a query executada é bastante mais complexa.

Listagem 2.4: Teste MySQL - Função de agregação

```
1 def group_by_10m(self, time_field, agg_field):
2     self.cur.execute("""
3         SELECT
```

```
4               {} DIV 600 * 600 AS by_time,
5               sum({}) as main_agg
6           FROM {}
7           GROUP BY by_time
8       """.format(time_field, agg_field, self.table_name))
9       return self.cur.fetchall()
```

De modo a conseguir agrupar a informação em intervalos de 10 minutos usando a linguagem SQL, cada valor de *timestamp* irá ser dividido, usando a divisão inteira, por 600 (número de segundos em 10 minutos), sendo que de seguida será multiplicado novamente por 600. Com esta lógica de operações, todos os eventos registados no mesmo período de 10 minutos irão obter o mesmo valor para o campo *timestamp*. Seguidamente é indicado ao MySQL que agrupe registos por este valor de *timestamp* obtido e, para cada um desses grupos, é pedido que o sistema calcule a soma dos valores presentes no campo que é recebido no argumento *agg_field*.

2.2.3 MongoDB

Segundo a documentação oficial deste sistema[2], o MongoDB é um sistema de gestão de base de dados gratuito e *open-source* que se foca em escalabilidade e facilidade de desenvolvimento. Vários estudos comparativos, incluindo o estudo em [7] analisado na secção anterior, indicam ainda que este sistema, bem como outros sistemas do mesmo tipo (NoSQL), têm ainda a vantagem de ter um melhor desempenho em operações de escrita e leitura para operações numa só coleção de dados. Foi escolhido o sistema MongoDB pois, uma vez que é considerado um dos sistemas NoSQL mais usados no mercado[3], existe mais informação online disponível que, juntamente com uma documentação bastante completa e organizada, facilita o desenvolvimento destes testes.

Para testar as operações no sistema MongoDB foi também usada uma biblioteca Python para auxiliar a comunicação com este sistema. A biblioteca usada foi a biblioteca *PyMongo* que já foi apresentada anteriormente.

Inserção e Leitura

Tal como referido na documentação, com o sistema MongoDB, é bastante simples de implementar as operações de leitura e escrita. Para cada uma destas operações, a sua implementação baseia-se em usar a função respetiva, disponível no objeto de coleção inicializado anteriormente. De seguida será apresentado um excerto de código que demonstra como estas funções foram implementadas.

Listagem 2.5: Teste MongoDB - Funções de inserção e leitura

```
1 def insert(self, data):
2     self.col.insert_many(data)
```

[2] https://docs.mongodb.com/manual/
[3] https://db-engines.com/en/system/Couchbase%3BMongoDB

```
3
4 def read_all(self):
5     return self.col.find()
```

A função *insert*, usada para realizar a inserção de dados, recebe como argumento, uma lista de objetos de registos, que irá passar directamente para a função *insert_many* do objeto de coleção do MongoDB. Para realizar a operação de leitura de todos os dados da coleção, é usada a função *find* do objeto de gestão da coleção. Esta função poderá receber argumentos, como filtros ou outras opções de procura, no entanto, quando executada sem argumento, esta função irá devolver todos os elementos.

Agregação

A biblioteca *PyMongo* fornece também funções para executar operações de agregação, como é o caso da função *aggregate*. Esta função irá devolver os resultados agrupados seguindo as regras passadas por argumento. Ao contrário das funções analisadas anteriormente, esta função requer como argumento um objeto bastante complexo para definir as regras de agregação.

Listagem 2.6: Teste MongoDB - Função de agregação

```
1  def group_by_10m(self, time_field, agg_field):
2      return self.col.aggregate([{
3          "$group": {
4              "_id": {
5                  "by_time": {
6                      "$multiply": [{"$divide": ["$"+time_field, 600]}, 600}]
7              },
8              "main_agg": { "$sum": "$"+agg_field }
9          }
10     }])
```

O excerto de código anterior demonstra como foram implementadas essas regras. A lógica usada é igual à usada para testar o mesmo tipo de operações no sistema MySQL, o valor do campo de tempo será dividido, usando a divisão inteira, por 600 e posteriormente multiplicado pelo mesmo valor. Os registos serão agrupados pelo valor resultante destas operações, o que significa que cada grupo inclui registos do mesmo intervalo de 10 minutos. Por cada um destes grupos será calculada a soma dos valores do campo cujo nome é passado pelo argumento *agg_field*, que, para estes testes, será sempre o campo que guarda o tempo de processamento. Por fim, o resultado desta função será devolvido pela função principal.

2.2.4 Elasticsearch

Embora não sendo considerado uma base de dados, Elasticsearch, um motor de busca, implementa uma interface REST que permite realizar as operações CRUD habituais para

criar, ler, atualizar ou apagar dados, bem como diversos tipos de operações de procura sobre os dados. Segundo a documentação oficial, o sistema Elasticsearch fornece uma forma eficiente de guardar informação, de forma a que seja possível realizar operações de leitura e agregação de uma forma mais rápida [4].

Mais uma vez, será usada uma outra biblioteca Python, para ajudar a comunicação com o sistema de armazenamento. No caso do sistema Elasticsearch, será usada a biblioteca *elasticsearch-py*, que também já foi apresentada anteriormente.

Inserção

Para implementar a função de inserção de dados, foi usado um objeto *helpers* fornecido pela biblioteca *elasticsearch-py* que facilita a implementação de várias funções, nomeadamente, facilita o processo de inserção de vários elementos em simultâneo. Para usar esta função é necessário realizar um pré-processamento simples dos dados recebidos por argumento. Para além do próprio objeto a guardar, para cada uma das entradas na lista, o Elasticsearch necessita que se defina o índice onde o elemento será inserido, bem como uma indicação do tipo de elemento. Depois de adicionada a informação necessária, será então iniciado o processo de inserção.

Listagem 2.7: Teste Elasticsearch - Função de inserção

```
def insert(self, data):
    for i in range(0, len(data)):
        data[i] = {
            "_index": self.index,
            "_type": self.data_type,
            "_source": data[i]
        }
    helpers.bulk(self.es, data)
```

Leitura

No caso do Elasticsearch, a implementação das funções para testes de leitura foi um pouco mais complexa. Este sistema usa paginação de resultados, assim, foi implementado um ciclo e, usando algumas funções auxiliares, iteradas todas as páginas de modo a obter todos os registos.

Listagem 2.8: Teste Elasticsearch - Função de leitura

```
def read_all(self):
    logs = []
    res = self.read_all_start()
    while res is not None:
        logs.extend(res["hits"])
        res = self.read_all_next(res["scroll_id"])
    return logs
```

O processo de leitura começa com a função *read_all_start*, esta função é responsável por definir as regras da procura que, para este caso, apenas definem que se pretende obter todos os registos.

Listagem 2.9: Teste Elasticsearch - Função auxiliar para iniciar a leitura

```
def read_all_start(self):
    result = self.es.search(index=self.index, size=10000, scroll="2m", body={
        "query" : {
            "match_all" : {}
        }
    })
    sid = result["_scroll_id"]
    hits = result["hits"]["hits"]
    scroll_size = len(hits)

    return {"scroll_id": sid, "hits": hits} if scroll_size > 0 else None
```

Na função *read_all_start*, as regras de procura são passadas pelo parâmetro *body* da função *search* do objeto da classe Elasticsearch. Para além deste argumento, terá ainda de ser indicado: o índice onde se pretende procurar a informação, usando o argumento *index*; a duração do identificador de paginação, usando o argumento *scroll*; por fim, o número de resultados por cada página, usando o argumento *size*. Quanto ao argumento *scroll*, o valor de dois minutos usado, permite obter toda a informação pretendida sem que haja problemas de desempenho. Quanto ao argumento *size*, foram realizados alguns testes de modo a tentar determinar o melhor valor a usar.

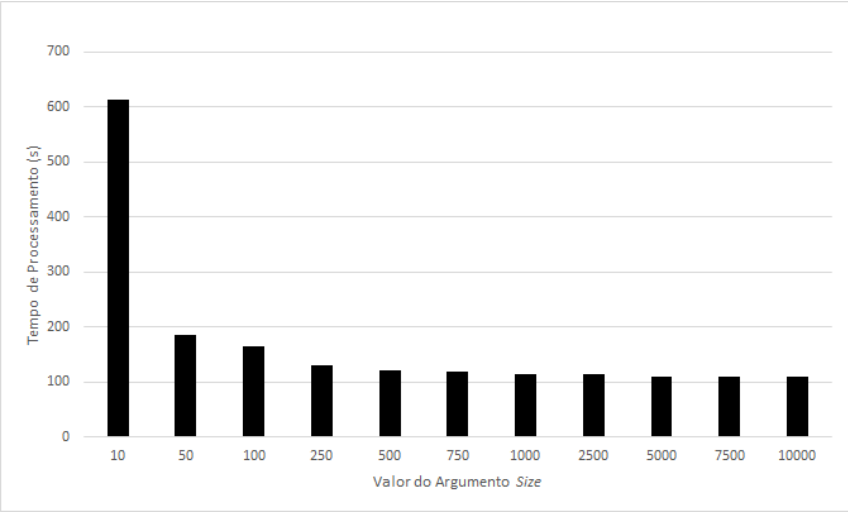


Figura 2.4: Teste de valores para o argumento *size* da função de procura do Elasticsearch

Na figura 2.4 encontram-se os resultados dos testes de leitura realizados com diferentes valores para o argumento *size*. Nos casos testados verificou-se que o tempo de processamento das operações de leitura decresce com o aumento do número de resultados pedidos, desse modo, foi escolhido o valor 10000 para o argumento *size*, por ser o

maior aceite pela função. Este resultado é, no entanto, relativo às condições em que o sistema foi testado. Foram testados outros casos em que o sistema Elasticsearch estava a ser executado num sistema com menos memória disponível, o que acabou por atrasar as operações de pesquisa quando o argumento *size* ultrapassava um certo valor.

Depois de indicados todos os argumentos, já é possível iniciar a obtenção de dados. Após executar a função *search*, para além de serem devolvidos os resultados, serão devolvidos também um identificador de paginação e o tamanho do resultado. Caso se verifique que ainda existam resultados, será devolvido um objeto com a informação dos resultados recebidos e do identificador de paginação para obtenção dos próximos resultados.

Listagem 2.10: Teste Elasticsearch - Função auxiliar para continuação da leitura

```
1 def read_all_next(self, scroll_id):
2     result = self.es.scroll(scroll_id=scroll_id, scroll="2m")
3     sid = result["_scroll_id"]
4     hits = result["hits"]["hits"]
5     scroll_size = len(hits)
6
7     return {"scroll_id": sid, "hits": hits} if scroll_size > 0 else None
```

Esta função *read_all_next* tem um funcionamento semelhante à função *read_all_start*, a única diferença é que esta usa uma função *scroll* para obter os resultados relativos a um certo identificador de paginação. De modo a poder identificar os resultados que se pretendem obter, é passado como argumento o identificador de paginação, que também será passado como argumento para a função *scroll*. Para além deste argumento só será necessário passar o argumento *scroll*, para indicar que se pretende continuar a obter resultados de forma paginada. Com apenas estes dois argumentos, o sistema terá toda a informação necessária para identificar os elementos que terá de devolver. O resultado desta função tem a mesma estrutura da função *search* usada anteriormente, desta forma, o restante código irá também extrair os resultados obtidos e o próximo identificador de paginação. O ciclo da função principal *read_all* irá terminar assim que se receber uma resposta vazia tanto na função *search* como na função *scroll*. Nestes casos, as funções auxiliares estão programadas para devolver também um resultado vazio, sendo que, o ciclo principal é terminado assim que um destes resultados seja verificado.

Agregação

Ao contrário dos sistemas analisados anteriormente, o sistema Elasticsearch permite facilmente definir um período de tempo para o agrupamento de dados, não sendo necessário recorrer às operações de divisão inteira e multiplicação.

Listagem 2.11: Teste Elasticsearch - Função de agregação

```
1 def group_by_10m(self, time_field, agg_field):
2     result = self.es.search(index=self.index, body={
3         "size": 0,
```

```
4         "aggs": {
5             "by_time": {
6                 "date_histogram": {
7                     "interval": "10m",
8                     "field": time_field,
9                     "format": "epoch_millis"
10                },
11                "aggs": {
12                    "main_agg": {
13                        "sum": {
14                            "field": agg_field
15                        }
16                    }
17                }
18            }
19        }
20    })
21    return result["aggregations"]["by_time"]["buckets"]
```

Para implementar operações de agregação é novamente usada a função *search* do objeto de classe Elasticsearch. Esta função recebe o nome do índice, onde se pretende procurar a informação, no argumento *index*. O argumento *body* permite definir todas as regras necessárias para as operações de agregação. Embora extenso, o objeto de configuração das operações de pesquisa, é relativamente simples. O campo *size* com valor a zero indica que se pretende receber um número ilimitado de resultados. O campo *aggs* é usado para definir as regras da agregação, sendo que o seu valor é um outro objeto. Neste objeto é definido um nome para a agregação, neste caso *by_time*, do tipo *date_histogram*. Este tipo de agregação permite facilmente definir regras para agrupamento de resultados por tempo, para tal, basta indicar o intervalo pretendido, usando o campo *interval*, o campo de tempo a analisar, usando o campo *field* e, finalmente, o formato esperado, usando o campo *format*, que, neste caso, se pretende que seja um valor de tempo em milissegundos. Ainda no campo *by_time*, é definida a operação a executar para cada grupo. Neste caso é definido que seja executada uma soma, definido pelo campo *sum*, dos valores do campo cujo nome é recebido no argumento *agg_field* da função. Depois de executada a procura, os seus resultados são extraídos do objeto devolvido pela função *search*, e devolvidos pela função principal.

2.2.5 Resultados

Usando as funções analisadas anteriormente, foram realizados vários testes, de modo a comparar o desempenho dos diferentes sistemas. Para a execução destes testes, foi implementada uma solução semelhante à solução final, com algumas alterações para permitir a medição de tempos de execução em vários pontos do código. Para além do tempo de execução em cada sistema de armazenamento, foi ainda medido o tempo de execução total, uma vez que, tanto o sistema MongoDB como o sistema MySQL, necessitam de um

pré-processamento adicional para transformar o valor de data num valor inteiro, para ser usado nas operações de divisão e multiplicação nas operações de agregação.

Todos os testes realizados e apresentados de seguida, foram realizados nas condições já indicadas anteriormente, e para o mesmo conjunto de dados. Para cada uma das operações analisadas, foram executados cinco testes e os resultados apresentados serão a média dos valores obtidos.

Inserção

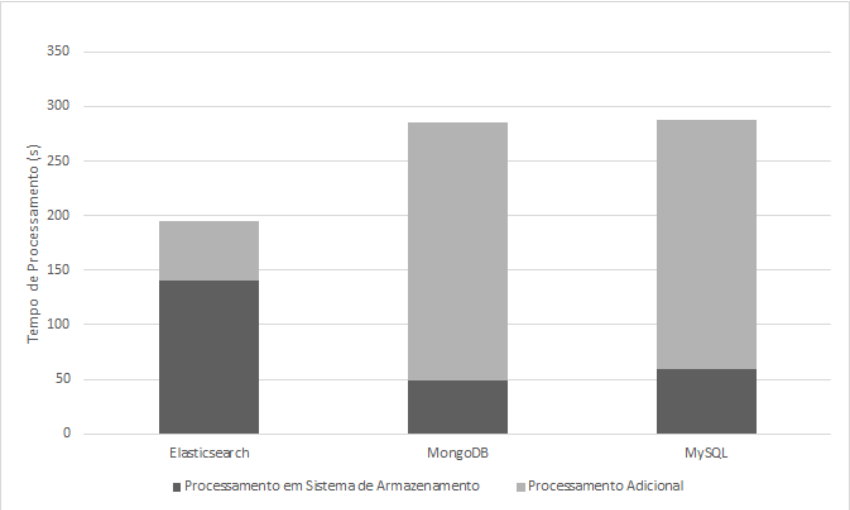


Figura 2.5: Resultados dos Testes de Inserção

A figura 2.5 apresenta os resultados obtidos para os testes de inserção efetuados. Analisando esta informação apresentada nesta figura observa-se que o sistema Elasticsearch é bastante mais lento para este tipo de operações do que os restantes sistemas analisados. De modo a tentar perceber esta diferença no desempenho foi realizada uma investigação acerca de como o Elasticsearch realiza as operações de inserção. Tal como referido na documentação, para definir cada elemento a ser inserido numa operação *bulk*, no corpo do pedido, é necessário indicar o índice onde esse elemento irá ser inserido, bem como o seu tipo[3]. Ao consultar a documentação da biblioteca usada, esta indica que foram criadas funções auxiliares para ajudar com este tipo de operações, visto que, os requisitos para especificação do formato, bem como outros fatores, poderão dificultar a implementação quando implementadas diretamente sobre a API do Elasticsearch[5]. Com esta informação, juntamente com o resultado de alguns testes usando versões de mais baixo nível destas operações, foi possível verificar que maior parte do tempo de processamento é usado para construir o pedido, sendo que, o tempo de processamento no sistema é bastante semelhante ao dos outros sistemas analisados.

Ainda que o sistema Elasticsearch possua um pior desempenho na inserção de elementos, no geral, para este caso específico, acaba por ter um desempenho melhor do que os outros sistemas. Uma das razões para a grande diferença no tempo de processamento

de código, é a necessidade de converter os valores de data para valores inteiros para todos os elementos a inserir. Pelo que foi possível verificar, esta operação é bastante complexa e atrasa bastante o pré-processamento dos registos.

Leitura

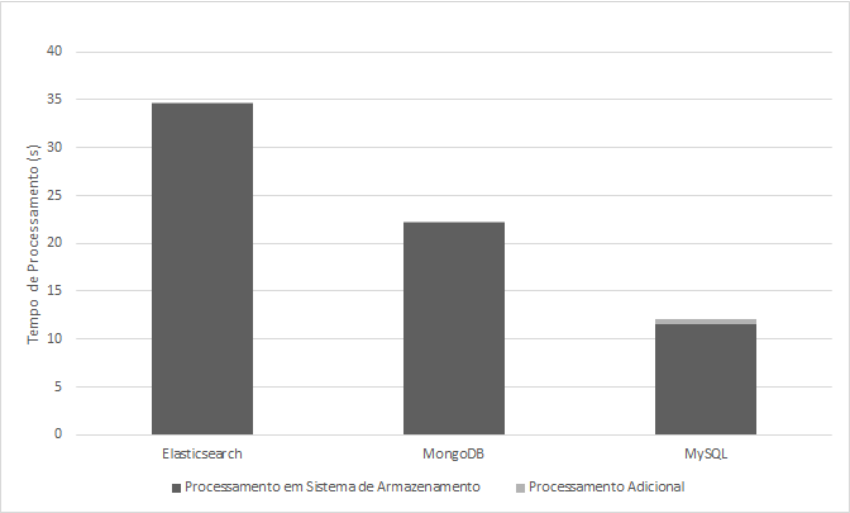


Figura 2.6: Resultados dos Testes de Leitura

Para testar o desempenho de leitura, foram testadas operações para obter todos os dados guardados pelos diferentes sistemas. Mais uma vez, analisando os resultados obtidos, representados na figura 2.6, verifica-se que o sistema Elasticsearch apresenta um pior desempenho quando comparado com os outros sistemas estudados. Quanto ao sistema MongoDB, ainda que possua um melhor desempenho da base de dados para operações de inserção, no caso das operações de leitura, verificou-se um pior desempenho que o sistema MySQL. Ainda que o tipo de operações não seja crucial para o funcionamento do sistema de monitorização, em certas ocasiões, poderá ser interessante ter acesso aos registos de modo a analisar toda a sua informação. Com este teste consegue-se perceber o custo da obtenção desses dados nos diferentes sistemas.

Agregação

Das operações analisadas, as operações de agregação são as mais relevantes no âmbito deste projeto. Um dos objetivos principais deste projeto é reduzir a quantidade de informação enviada para a camada de apresentação onde será apresentado o painel de controlo. De forma a reduzir a quantidade de informação será necessário usar operações deste género.

Ao analisar os resultados obtidos, representados em forma de gráfico na figura 2.7, é possível verificar que é neste tipo de operações que o sistema Elasticsearch tem claras vantagens de desempenho relativamente aos restantes sistemas. Embora o seu desempenho

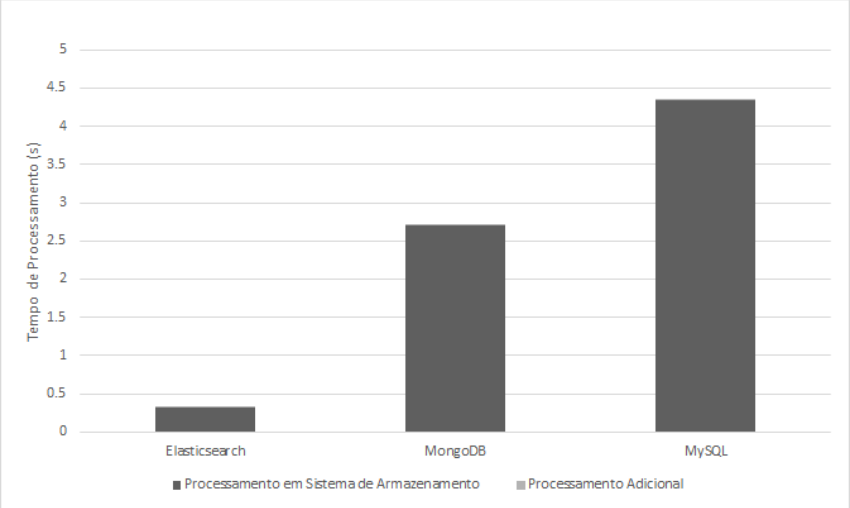


Figura 2.7: Resultados dos Testes de Agregação

não seja relativamente bom para operações de leitura simples, este sistema foi desenvolvido especialmente para este tipo de operações para auxiliar a análise de dados[4], apresentando portanto, neste caso, um desempenho muito superior ao dos restantes sistemas.

2.2.6 Conclusões

Com o estudo apresentado foi possível testar estes três sistemas de armazenamento, de uma forma mais específica às necessidades deste projeto. Após a análise destes resultados, bem como a análise da frequência com que estas operações serão usadas para efeitos de monitorização, concluiu-se que o sistema Elasticsearch é aquele que melhor se adequa ao projeto.

Em termos de operações, a solução final está desenvolvida para ler a informação necessária apenas uma única vez por cada ficheiro, deste modo, a maior parte de operações realizadas sobre o sistema serão de leitura. Como já referido várias vezes neste documento, devido às limitações de processamento da camada de apresentação, estas operações de leitura terão de devolver resultados agregados o mais rápido possível. O painel de controlo, implementado também durante esta dissertação, permite obter vários tipos de informação estatística sobre o conteúdo dos registos e ainda, criar e alterar filtros sobre a informação apresentada de uma forma interativa. Deste modo, o bom desempenho destas funções de agregação, é ainda mais importante, visto que se pretende que o utilizador consiga retirar conclusões acerca da informação, o mais rapidamente possível.

Para além do seu bom funcionamento e desempenho nas principais funções executadas durante as ações de monitorização e *debug*, o sistema Elasticsearch possui um bom ecossistema e várias ferramentas que facilitam imenso, tanto o desenvolvimento de código para processamento de registos, como também, a gestão de informação contida no sistema, e ainda, a configuração de painéis de controlo, quando usado o sistema Kibana.

Outros aspetos positivos deste usar este sistema são a facilidade de instalação e configuração, bem como a facilidade de configurar operações de agregação, algo que é percetível, tanto nos excertos de código apresentados, como também através de outras fontes[9].

2.3 Análise de Soluções de Apresentação

Durante este projeto de mestrado foram estudados dois sistemas para criação e configuração de painéis de controlo para ações de monitorização e *debug*. Estes sistemas são o sistema Grafana e o sistema Kibana. Ambos estes sistemas fornecem funcionalidades semelhantes para apresentação de informação, sendo que o fator de decisão para um destes sistemas foi a forma como se relacionam com as restantes partes da solução desenvolvida.

O sistema Grafana foi o sistema inicialmente usado, ainda antes do início deste projeto, a Quidgest implementou um painel de controlo usando este sistema, como forma de facilitar o processo de encontrar as causas de alguns erros reportados por clientes. Este sistema funciona relativamente bem com o sistema de armazenamento SQL usado na altura, sendo que o principal problema era a quantidade de informação que estava a ser processada.

Uma vez que este sistema já estava a ser usado anteriormente, após a escolha do sistema de armazenamento a usar, e depois de todo o processamento de registos implementado, foi implementado um painel de controlo simples usando também este sistema. Ainda que o Grafana suporte o sistema Elasticsearch, este suporte é bastante limitado, tendo sido encontradas algumas dificuldades, por exemplo, em criar tabelas para apresentar os registos de um determinado intervalo de tempo, ou também, definir opções mais avançadas para pesquisa.

O sistema Kibana foi uma alternativa ao Grafana proposta pela Quidgest. Este sistema foi bastante útil durante todo o desenvolvimento deste projeto. Para além de ferramentas para criar e editar painéis de controlo, o sistema Kibana permite facilmente gerir dados e coleções do sistema Elasticsearch, algo que foi crucial em todo o processo de teste, tanto do próprio sistema como do código para processamento de registos. Sendo o Kibana desenvolvido pela mesma equipa do Elasticsearch, a sua integração no Kibana é excelente, sendo que não foi encontrada neste sistema qualquer das limitações encontradas no sistema Grafana. Por estes motivos, foi adotado o sistema Kibana como o sistema para implementação da apresentação visual dos dados.

2.4 Tecnologias e Ferramentas Utilizadas

Para desenvolver esta solução foi necessário recorrer ao uso de diversas tecnologias e ferramentas, algumas das quais já foram indicadas anteriormente, no entanto, estas, e as restantes, serão apresentadas mais detalhadamente nas próximas subsecções.

2.4.1 GENIO

GENIO[4] é uma plataforma de desenvolvimento *NoCode* (sem código), baseada em modelos e padrões de sistemas e desenvolvida internamente pela Quidgest. Esta plataforma, tal como outras do mesmo tipo, possui vantagens relativamente a outras formas de desenvolvimento de sistemas, tais como: facilidade de desenvolvimento, sendo que qualquer pessoa, mesmo que não possua qualquer formação ou conhecimentos na área de engenharia de software, pode participar, de diferentes formas, no desenvolvimento de aplicações; rapidez de desenvolvimento, uma vez que o desenvolvimento é baseado em modelos e padrões anteriormente desenvolvidos, estes já se encontram bem testados, devido a testes realizados no âmbito de outros projetos.

Apesar do trabalho a ser realizado não ter uma relação muito direta com esta plataforma, esta tem importância no âmbito deste projeto por ser a plataforma responsável pelo desenvolvimento da aplicação web estudada durante a realização desta dissertação.

2.4.2 CentOS

CentOS[5] é um sistema operativo baseado em Linux. Estes sistemas operativos são bastante utilizados em máquinas de servidores, principalmente devido aos seus menores requisitos de hardware, quando comparado com sistemas Windows. Relativamente ao CentOS, este sistema é especialmente desenvolvido para ambientes de produção. Fornecendo pacotes com versões de software mais estáveis, estes sistemas são menos sujeitos a erros. Adicionalmente, cada versão do CentOS é suportada até 10 anos, com várias atualizações de segurança e de suporte a novo hardware.

Apesar de a Quidgest desenvolver software para Windows (como indicado na subsecção anterior), para o desenvolvimento da solução deste projeto de mestrado é usado o sistema CentOS onde são instaladas as restantes ferramentas.

2.4.3 Grafana

Grafana [6] é uma plataforma especializada em monitorização que serve como um *front-end* para um painel de controlo de monitorização. Com a ajuda do Grafana é fácil apresentar dados de várias formas, visto que a plataforma suporta uma grande variedade de gráficos e outros tipos de visualização de dados. Outras funcionalidades bastante úteis são, por exemplo: a possibilidade de facilmente definir um intervalo de tempo e apresentar só informação que corresponda a esse intervalo; a possibilidade de gerar alertas simples para quando certas métricas atingem certos valores; suporte para múltiplos utilizadores e tipos de utilizador, sendo que cada um pode personalizar o seu painel de controlo da forma que lhe seja mais útil; controlo de acessos e possibilidade de ser integrado com a

[4] https://www.quidgest.pt/q_genioPT.asp
[5] https://www.centos.org/
[6] https://grafana.com/

autenticação Windows no domínio da empresa; software gratuito, livre e de código aberto, criando a possibilidade de se poder desenvolver ou usar extensões adicionais.

Esta ferramenta foi utilizada para implementar a apresentação dos dados na solução provisória desenvolvida pela Quidgest. Durante o processo de análise de soluções de apresentação descrito em 2.3, foi ainda usada em alguns testes, onde se concluiu que não seria a melhor ferramenta do tipo para ser usada no contexto deste projeto.

2.4.4 Kibana

Kibana [7] é uma outra plataforma que permite a criação de painéis de controlo para a monitorização. Este sistema implementa funcionalidades semelhantes ao Grafana, permitindo também criar painéis de controlo com diversos tipos de visualizações de dados. Adicionalmente, o Kibana tem a particularidade de ser desenvolvido pelos mesmos criadores do Elasticsearch, e, desse modo, inclui algumas ferramentas úteis para efetuar a gestão dos dados armazenados nesse sistema.

Esta ferramenta é uma alternativa ao Grafana que foi estudada durante o desenvolvimento deste projeto 2.3. Para além do estudo efetuado sobre várias formas de apresentar informação nesta plataforma, esta foi muito usada durante todo o período de desenvolvimento para explorar a informação guardada no sistema Elasticsearch, e ainda, para gerir coleções de dados ou índices. Finalmente, como também já foi apresentado, Kibana foi a ferramenta escolhida para criar as ferramentas de apresentação da informação obtida dos registos processados.

2.4.5 MySQL

MySQL [8] é um sistema de gestão de base de dados que segue o modelo relacional. Este sistema é autualmente o sistema mais utilizado no mercado[9]. Algumas vantagens deste sistema incluem a facilidade de instalação e configuração, facilidade de desenvolvimento utilizando a linguagem SQL e ainda, bom desempenho para casos onde os tipos da informação guardada possuam relações entre si[10].

Para a implementação da solução provisória, a Quidgest optou por usar o sistema Microsoft SQL Server[11] para realizar a gestão da base de dados. Este sistema foi escolhido devido à familiaridade que a Quidgest tem com o mesmo, e também ao facto do sistema operativo usado ser o sistema Windows. Para o desenvolvimento deste projeto, a Quidgest pretendia que a nova solução fosse implementada sobre o sistema operativo CentOS, este sistema, sendo baseado em linux, não é compatível com o Microsoft SQL Server. Deste modo, decidiu-se considerar o sistema MySQL como uma das potenciais soluções de

[7] https://www.elastic.co/products/kibana
[8] https://www.mysql.com/
[9] https://www.edumobile.org/database/the-most-popular-databases-2019/
[10] https://www.geekboots.com/story/why-mysql-so-popular-in-web
[11] https://www.microsoft.com/pt-pt/sql-server/sql-server-2019

armazenamento a adotar na implementação da solução final, devido à sua similaridade com o sistema anterior e também por ser o sistema mais popular do tipo.

Tal como indicado em 2.2, este sistema não foi o escolhido para implementar a solução final, devido, principalmente, ao seu pior desempenho neste projeto e maior dificuldade em criar operações de agregação de dados, quando comparado com outras soluções estudadas.

2.4.6 MongoDB

MongoDB [12] é um sistema de gestão de bases de dados NoSQL conhecida pela sua melhor performance relativamente a outros SGBD, principalmente relativamente a SGBD relacionais. Possui um melhor desempenho tanto em escrita como leitura de informação para operações realizadas numa só tabela (ou coleção no MongoDB)[7]. De modo a armazenar a informação, MongoDB usa documentos e guarda a informação dos vários atributos num formato JSON.

Este sistema foi um dos escolhidos para o estudo sobre o armazenamento da informação a ser processada. As razões para essa escolha já foram apresentadas em 2.1.2 e incluem, principalmente, o seu bom desempenho em operações de escrita e leitura [7]. Ainda que MongoDB apresente algumas vantagens, com o estudo apresentado em 2.2, concluiu-se que, mesmo assim, não seria a melhor opção para as necessidades deste projeto.

2.4.7 Elasticsearch

Elasticsearch [13] é uma outra opção a considerar para o armazenamento de dados. Elasticsearch não é oficialmente considerado um sistema de gestão de base de dados, mas sim um motor de busca, no entanto, possui uma interface que implementa funções semelhantes. Segundo os testes realizados e apresentados em 2.2, esta solução, relativamente a MongoDB, possui uma menor velocidade de escrita e leitura para elementos numa só coleção, ainda assim, conseguiu-se verificar que este sistema possui uma grande vantagem no que diz respeito ao desempenho de operações de agregação. Uma vez que estas operações de agregação são operações bastante importantes para o ambiente de monitorização, especialmente para agrupar dados de modo a enviar a menor quantidade informação possível para o painel de controlo, esta foi a solução de armazenamento escolhida para ser usada na solução final.

2.4.8 Python

Python [14] é uma linguagem de programação de alto nível, interpretada por um interpretador específico que possui versões para os principais sistemas operativos (Windows,

[12] https://www.mongodb.com/
[13] https://www.elastic.co/products/elasticsearch
[14] https://www.python.org/

MacOS e Linux), que se foca especialmente na facilidade de desenvolvimento e leitura do código resultante.

Esta foi a linguagem de programação escolhida para o desenvolvimento deste projeto devido à grande rapidez e facilidade de desenvolvimento e manutenção de código, bem como a legibilidade do código desenvolvido que permite facilmente perceber o que foi implementado, como foi implementado e o seu funcionamento no geral. Outra grande vantagem do ecossistema Python é o grande número de bibliotecas disponíveis que implementam interfaces para ligação com outro tipo de sistemas e ferramentas. Entre outras bibliotecas usadas, destacam-se as seguintes:

- **PyMongo**[15]: A biblioteca recomendada pelo MongoDB para a comunicação entre o seu sistema de gestão de bases de dados e o código Python, para tal, fornece várias funções que implementam as funcionalidades do sistema. Este fator, juntamente com a quantidade e qualidade da documentação disponível online facilitam bastante o desenvolvimento de código.

- **Elasticsearch-Py**[16]: A biblioteca oficial para a comunicação entre o sistema Elasticsearch e o código Python. Existe uma outra biblioteca, implementada por cima desta, que fornece uma API mais simples, no entanto, Elasticsearch-Py fornece uma camada de abstração suficiente para facilitar o desenvolvimento, mas permitindo ao utilizador o controlo total sobre as funcionalidades do sistema Elasticsearch. Também para esta biblioteca, existe bastante documentação oficial online.

2.5 Resumo

Todo o trabalho analisado e apresentado neste capítulo influenciou, de algum modo, a estrutura e funcionalidades da solução proposta neste documento.

O artigo [1] apresenta uma possível solução para a monitorização de um programa. A utilização de um processador para o processamento do programa e um outro, independente, para o processamento de registos de eventos, permite um melhor desempenho de ambos os sistemas. Embora seja bastante útil nos casos estudados no artigo, esta solução tem algumas limitações relativamente ao caso de estudo deste projeto de dissertação. A solução apresentada no artigo necessita de hardware específico para capturar os eventos, enquanto que, para a solução desenvolvida neste projeto, pretende-se apenas que sejam usadas as funcionalidades de registo de eventos atualmente presentes nos diferentes subsistemas a analisar. Outro problema relaciona-se com o nível em que é efetuada a monitorização. Vários subsistemas podem não ter código aberto, ou ser demasiado complexos. Nesses casos, não é particularmente útil saber, por exemplo, que endereços de memória foram usados ou a instrução de input/output executada. Registos de mais alto

[15] https://api.mongodb.com/python/current/
[16] https://elasticsearch-py.readthedocs.io/en/master/

nível (como os presentes nos ficheiros a analisar neste projeto) podem conter informação resumida acerca de ações executadas no sistema. Por norma, essa informação é suficiente para perceber se alguma função de um subsistema não foi executada da melhor forma, ou se gerou algum erro interno. Durante esta dissertação pretende-se adaptar esta solução para que possa ser usada em mais alto nível.

No artigo [7] é efetuada uma comparação entre vários sistemas de gestão de bases de dados, onde, após a observação dos seus resultados, se conclui que MongoDB e Couchbase parecem ser as melhores opções para armazenar os registos de eventos. Os resultados obtidos neste trabalho para as duas soluções são bastante semelhantes, deste modo, uma vez que MongoDB possui uma melhor documentação, foi escolhido este sistema para representar soluções do tipo NoSQL num novo teste realizado durante este projeto. Este novo teste estendeu a comparação entre soluções do tipo SQL e NoSQL para incluir também um outro tipo solução de armazenamento presente no sistema Elasticsearch. O trabalho realizado, bem como os seus resultados foram apresentados em 2.2. Nesta secção foi indicado que, no final dos testes e análise, foi determinado que Elasticsearch é a tecnologia que melhor se adequa ao projeto.

Em [6] é estudado um sistema de monitorização que se baseia em registos de vários sensores para criar um gráfico que permite analisar a evolução destes valores ao longo do tempo. De certa forma, procura-se implementar algo semelhante na solução apresentada nesta dissertação. No entanto, ao usar as funcionalidades dos softwares de painel de controlo, é possível melhorar este tipo de gráficos tornando-os interativos e ainda apresentar informação adicional usando vários outros tipos de gráficos e tabelas. Adicionalmente, este artigo da NASA também refere a possibilidade de incluir algoritmos de aprendizagem automática para a deteção de anomalias. Este tipo de soluções são interessantes porque podem facilitar imenso todo o processo de deteção de falhas.Ainda que este projeto não se foque em implementar uma destas soluções em concreto, foi desenvolvido com a preocupação de possibilitar que estas sejam facilmente adotadas no futuro.

Na secção 2.3 é ainda apresentada a explicação por detrás da escolha do Kibana para implementar a camada de apresentação do projeto. Escolheu-se o Kibana porque, para além da facilidade em criar e gerir painéis de controlo, este sistema também se integra muito bem com o sistema de armazenamento escolhido, o Elasticsearch.

Com esta dissertação procurou-se juntar os aspetos positivos de todos estes trabalhos, adaptá-los ao contexto do projeto, e usá-los como ponto de partida para criar uma solução mais completa (baseada na solução já existente) que satisfaça as necessidades atuais de monitorização de sistemas na Quidgest.

Capítulo 3

Implementação

3.1 Estrutura da Solução

Nesta secção será apresentada a estrutura da solução desenvolvida. Esta secção serve também como introdução ao desenvolvimento apresentado posteriormente neste documento. Com a informação presente neste capítulo o leitor deverá conseguir compreender mais facilmente como é que todas as partes do projeto interagem de modo a que a solução final funcione corretamente.

3.1.1 Estrutura Global

A estrutura global da solução desenvolvida é bastante semelhante à estrutura implementada previamente pela Quidgest. Esta estrutura já foi descrita e está representada visualmente através do esquema na figura 2.3. A figura 3.1 representa a estrutura da nova solução. Esta nova estrutura é bastante semelhante à anterior e, de modo a facilitar a comparação entre as duas, esta nova figura irá representar com linhas tracejadas, as partes do sistema que foram adicionadas ou alteradas.

Nesta figura é possível ver que se mantém a separação dos sistemas. Tal como referido anteriormente, esta separação de sistemas, por um lado permite o desenvolvimento de uma solução de monitorização completamente independente da estrutura do sistema principal (dependendo apenas na estrutura dos ficheiros de registo de eventos) e que, por outro lado, melhora o desempenho do sistema no geral (uma vez que elimina a necessidade de partilhar os recursos de uma única máquina, melhorando assim o desempenho de todas as partes do sistema). Com esta separação de sistemas, o sistema da aplicação pode ser visto apenas como um bloco que gera ficheiros de registos, ignorando assim a estrutura e software concretos usados para suportar a aplicação web, o que permite criar uma solução de monitorização relativamente independente desses fatores.

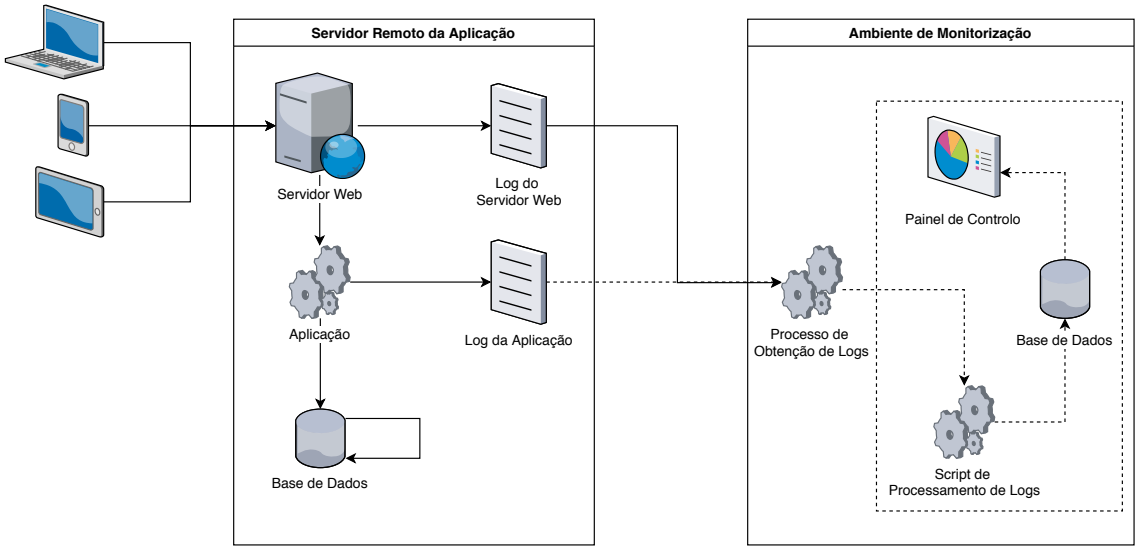


Figura 3.1: Estrutura global do sistema

Do lado do ambiente de monitorização foi quase tudo repensado. Na figura 3.1 está representada, dentro de uma caixa a tracejado, a parte do sistema que sofreu alterações. Um dos objetivos deste projeto é analisar as soluções implementadas previamente e compará-las com outras soluções alternativas do mesmo tipo. Deste modo, pretende-se, implementar uma estrutura usando uma combinação de sistemas, que permita obter o melhor desempenho possível nas ações de monitorização e de resolução de erros. Destes sistemas, o que mais influencia o desempenho da solução final é o sistema de base de dados, sendo que foi necessário substituir o sistema Microsoft SQL Server que estava a ser utilizado pelo sistema Elasticsearch que provou ter um melhor desempenho (como se viu em 2.2). Outro sistema que foi alterado foi o sistema responsável pela apresentação do painel de controlo. O sistema Grafana que estava a ser usado anteriormente, foi substituído pelo sistema Kibana, pelas razões também já vistas anteriormente em 2.3.

Uma das maiores alterações à estrutura do sistema foi a introdução de um processo intermédio entre a obtenção dos registos e a sua inserção da base de dados que será descrito de seguida.

3.1.2 Estrutura da Solução de Processamento de Registos

Um dos problemas que a solução de monitorização e resolução de erros que a Quidgest implementou relaciona-se com a forma como a informação é guardada nas bases de dados locais. Por um lado, a quantidade de informação guardada é excessiva, uma vez que toda a informação presente nos registos é guardada, independentemente da sua importância para as ações de monitorização e de resolução de erros. Por outro lado, alguma da informação não estava a ser guardada com a a estrutura correta, principalmente nos campos que guardam os endereços invocados pelos clientes. Quando o formato do endereço não era o formato específico esperado, a informação processada não era corretamente distribuída

pelos diferentes campos.

Para resolver os problemas referidos anteriormente (e possibilitar a implementação de operações de pré-processamento de dados) foram criados alguns scripts Python responsáveis pela análise dos ficheiros de registos, por executar todas as funções de pré-processamento e ainda por enviar os dados processados para o sistema de armazenamento para ficarem armazenados de uma forma mais permanente. Ainda que estes scripts criem uma necessidade de efetuar algum processamento adicional (quando comparando com a solução anterior), considera-se que a sua introdução e utilização é vantajosa, visto que, para cada ficheiro de registos, este processamento só é executado uma vez quando a informação é inserida.

Uma preocupação adicional presente durante todo o processo de desenvolvimento de toda a solução de monitorização e resolução de erros é a de implementar esta solução de uma forma a facilitar a sua alteração, configuração ou mesmo adicionar novos tipos de sistemas no futuro. Adicionalmente, não se pretende que esta solução aqui apresentada funcione exclusivamente para o sistema concreto que se está a estudar, esta solução deve poder ser flexível o suficiente para permitir a sua utilização em outros casos semelhantes. Por estes motivos, o desenvolvimento da solução de processamento de ficheiros de registos foi realizado usando o paradigma de programação orientada a objetos, dividindo a solução em várias partes, responsáveis por diferentes acções. A figura 3.2 representa esquematicamente as diferentes partes da solução de processamento, e ainda, que ficheiros ou classes se encontram em cada uma destas partes. Nesta figura estão representados quatro ficheiros referentes aos ficheiros com código Python desenvolvidos para o processamento dos ficheiros de registos de eventos.

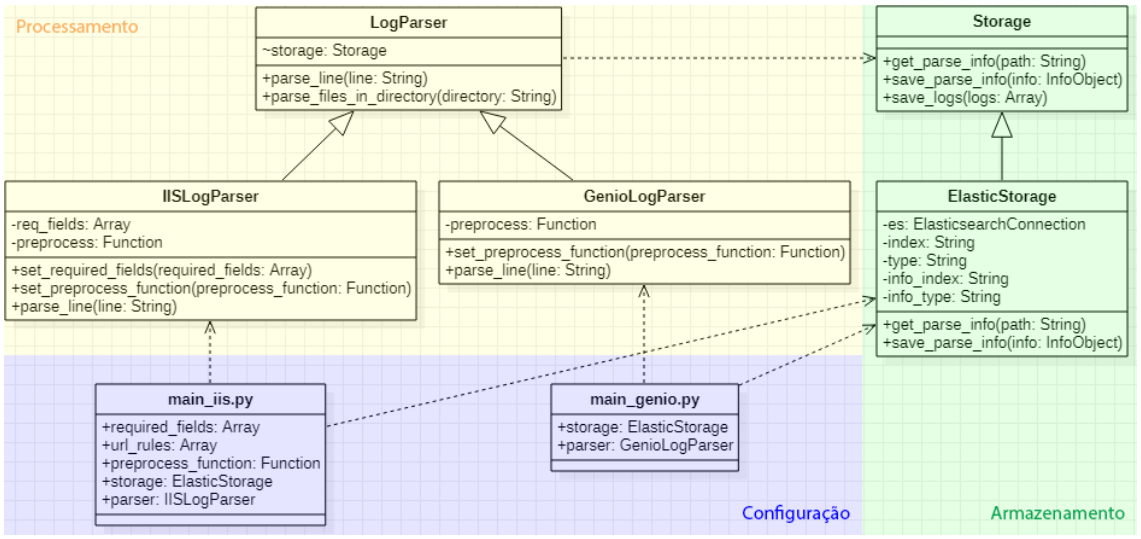


Figura 3.2: Estrutura da solução de processamento de registos

3.1.2.1 Configuração

A parte de configuração desta solução permite a definição de várias regras que deverão ser usadas durante o processamento dos ficheiros, bem como a definição das funções de pré-processamento. Depois de implementada toda a configuração necessária, os ficheiros de configuração irão inicializar o processamento em concreto, para tal, estes ficheiros incluem os ficheiros da fase seguinte, a fase de processamento. Embora que, tal como indicado na figura 3.2, só exista um ficheiro de configuração, o ficheiro *main.py*, é possível que sejam criados mais ficheiros deste tipo, por exemplo, se for necessário separar o processamento de ficheiros de registo dos diferentes sistemas, neste caso o servidor IIS e o sistema GENIO da Quidgest. No caso específico deste projeto optou-se por manter apenas um ficheiro, de modo a facilitar a inicialização de todo o processamento.

3.1.2.2 Processamento

A parte de processamento é onde é implementado o processamento dos ficheiros de registos. Para cada sistema que se pretenda analisar será necessário implementar uma classe específica para processar a sintaxe específica dos ficheiros de registos que o sistema cria. Estas classes possuem uma estrutura fixa, sendo que todas elas têm de implementar um conjunto de funções específicas que servirão como interface para os ficheiros na parte da configuração. As classes de processamento recebem as regras e as funções de pré-processamento, processam todas as linhas dos ficheiros, separam a informação de cada registo e, baseando-se nas regras e funções definidas, constroem uma estrutura de dados útil para posteriormente ser usada nos painéis de monitorização. Depois de construir essa estrutura, as classes de processamento usam as classes de armazenamento para enviar a informação para os respetivos sistemas de armazenamento. Neste projeto específico foram analisados ficheiros de registos de dois subsistemas, o servidor web IIS e a aplicação desenvolvida pela Quidgest, o GENIO, desse modo, existem duas classes para realizar o processamento destes ficheiros.

3.1.2.3 Armazenamento

A parte de armazenamento desta solução permite possibilitar o uso de outros tipos de sistemas de armazenamento que não aquele que foi escolhido para este projeto. Neste documento será apresentado o estudo efetuado a sistemas de armazenamento, desse estudo concluiu-se que o sistema Elasticsearch era o que melhor se adequava ao projeto, desse modo, foi implementada a classe de armazenamento para a comunicação com esse sistema. Ainda que o estudo efetuado indique que Elasticsearch é o melhor sistema a usar para estes casos, poderão existir outros fatores que venham a impedir o seu uso em outros projetos, assim, caso se pretenda usar outro sistema, pode ser implementada outra classe de armazenamento, com a mesma estrutura da que já foi implementada, que envie a informação para o sistema pretendido. Esta separação das soluções de armazenamento

foi especialmente útil para efetuar alguns testes no estudo referido anteriormente uma vez que foi necessário testar vários sistemas diferentes.

3.1.3 Estrutura da Solução de Processamento para Testes

Durante a fase de análise de soluções de armazenamento, apresentada em 2.2, a estrutura usada para o processamento foi semelhante à estrutura apresentada anteriormente, no entanto, existem algumas diferenças. A figura 3.3 ilustra esta nova estrutura.

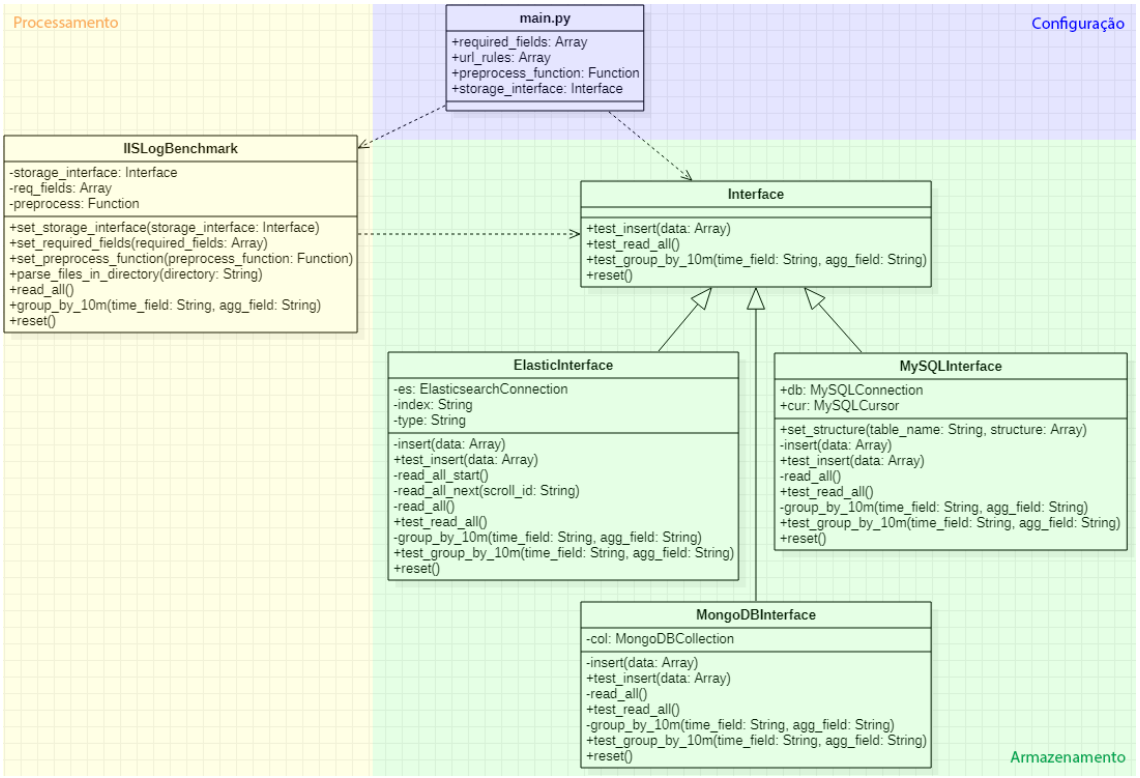


Figura 3.3: Estrutura da solução de processamento para testes

As maiores diferenças desta estrutura, quando comparada com a apresentada na secção anterior, são o facto de existirem mais dois ficheiros para o armazenamento e a utilização de apenas os ficheiros do sistema IIS, processados na classe IISBenchmark. As classes usadas nesta fase, embora sejam baseadas nas apresentadas na figura 3.2, são bastante diferentes. Estas novas classes foram implementadas com o objetivo de fornecer informação acerca de quanto tempo as acções de escrita e leitura demoram a ser processadas pelos diferentes sistemas de armazenamento, algo que não é feito no processamento normal dos registos.

3.2 Análise de Registos de Eventos

Nesta secção irá ser apresentado o trabalho executado em termos de análise dos registos de eventos. Após realizada a análise a alguns ficheiros de registos de eventos, a informação

obtida irá ser usada para configurar o processamento de registos. Esta configuração será implementada no script de entrada *main.py*, onde serão definidas várias regras específicas para cada tipo de registos.

Visto que este sistema de monitorização é executado localmente, antes de se poder executar qualquer tipo de processamento de dados, é necessário transferir esses dados, do ambiente de produção para a máquina local. Existem várias opções para efetuar a transferência dos registos, no entanto, para o propósito deste projeto, será apenas assumido que existe algo que irá transferir esses registos para uma diretoria local no ambiente de monitorização, sendo que não se irá entrar em mais detalhe sobre soluções em específico.

3.2.1 Registos do Servidor IIS

IIS é um servidor WEB que recebe os pedidos dos clientes e os encaminha para serem processados pela aplicação. Os registos de eventos deste subsistema, têm elevada importância para a monitorização do sistema e deteção de anomalias. Como este servidor recebe e responde aos pedidos dos clientes, o fluxo de processamento e de dados, inicia-se e termina neste servidor. Deste modo, todos os atrasos ou erros, que ocorram durante todo o processamento da aplicação, irão ser refletidos em atrasos ou erros também no servidor IIS, e consequentemente nos seus registos de eventos.

No caso da aplicação estudada, não existe apenas um servidor WEB. De modo a melhor distribuir a carga, existe um servidor de balanceamento que divide os pedidos entre duas máquinas (erp1 e erp2), sendo que cada uma processa uma instância do servidor IIS. Assim sendo, existem vários ficheiros de registos de eventos, gerados pelos diferentes servidores, no entanto, a sua estrutura mantém-se igual.

3.2.1.1 Estrutura dos Ficheiros dos Registos

Os ficheiros de registos contêm bastante informação, no entanto, alguma desta informação não é especialmente útil para obter informação sobre o estado do sistema. Cada um destes ficheiros contém vários registos, cada um contendo a informação do respetivo evento, dividida entre vários campos configuráveis no servidor web. Ainda que estes campos pudessem ser configurados de forma a que, apenas aqueles campos relativos à monitorização, seriam guardados, poderá haver a necessidade de registar informação adicional sobre os eventos para outros casos.

Cada ficheiro contém várias linhas, sendo que cada uma dessas linhas pode ser um registo de um evento ou um comentário. Cada linha de registo, descreve o evento através de informação separada por vários campos, a ordem e o significado desses campos são definidos previamente através de um comentário específico. Este tipo de comentário poderá aparecer várias vezes no mesmo ficheiro, o que significa uma possível alteração no formato dos registos guardados posteriormente. O excerto de código em 3.1 contém um exemplo de um destes comentários.

Listagem 3.1: Exemplo do comentário que define a estrutura dos registos no ficheiro

```
1 #Software: Microsoft Internet Information Services 8.5
2 #Version: 1.0
3 #Date: 2019-02-15 00:00:58
4 #Fields: date time s-sitename s-computername s-ip cs-method cs-uri-stem
5 cs-uri-query s-port cs-username c-ip cs-version cs(User-Agent) cs(Cookie)
6 cs(Referer) cs-host sc-status sc-substatus sc-win32-status sc-bytes cs-bytes
7 time-taken
```

A informação mais relevante deste comentário encontra-se na quarta linha. Nesta linha, encontram-se os nomes dos vários campos, separados por espaços, e na mesma ordem pela qual estes são apresentados nos registos dos eventos. Foi realizada uma análise destes campos e dos seus respetivos valores nos registos. Os resultados desta análise serão apresentados de seguida.

- **date**: Data em que o evento foi registado. Embora informativo, este campo ganha muito mais utilidade quando combinado com o campo de tempo, descrito de seguida;
- **time**: Tempo do dia em que o evento foi registado. Em conjunto com o campo de data, consegue-se obter toda a informação necessária para identificar exatamente em que ponto no tempo o evento foi registado;
- **s-sitename**: Nome do site configurado no servidor web. Cada um destes sites trata de um subconjunto dos pedidos dos clientes recebidos. No caso específico do servidor analisado neste projeto, existem 3 sites: W3SVC1, W3SVC2 e W3SVC3. W3SVC1 trata de todos os pedidos recebidos no porto 443, este é o porto predefinido para o tráfego web por canais seguros, e, uma vez que, neste servidor, o tráfego é redirecionado para este tipo de canais, os pedidos aqui recebidos, correspondem à maior parte dos pedidos. W3SVC2 trata de pedidos recebidos no porto 4444, sendo que este é um post configurado pela Quidgest para receber outro tipo de tráfego por Web Services. Por fim, W3SVC3 trata dos pedidos recebidos no porto 80, ainda que a predefinição seja a comunicação pelo porto 443, em alguns casos, o cliente pode tentar aceder ao porto 80, nestes casos, o servidor limita-se a redirecionar o cliente para o porto 443, de forma a continuar a comunicação. Em termos de métricas e informação para monitorização, este campo não fornece muita informação, desse modo, não será incluído na estrutura final dos registos guardados. Adicionalmente, o tráfego e registos analisados correspondem apenas ao site W3SVC1;
- **s-computername**: Nome da máquina que recebeu o pedido. No caso da Quidgest, existem dois servidores web que recebem pedidos, os nomes destas máquinas são erp1 e erp2. Em termos de monitorização, é importante guardar o nome do servidor, caso sejam detetadas anomalias, saber em que servidor ocorreram pode indicar possíveis problemas específicos com uma das máquinas;

- **s-ip**: IP do cliente que enviou o pedido. Este campo também será guardado na estrutura final, principalmente para casos de tentativa de intrusão, é interessante saber a sua origem, adicionalmente, essa informação pode ser usada para relacionar várias situações destas;

- **cs-method**: Método de http usado no pedido. Devido à sua importância para a monitorização, este campo será incluído na estrutura final dos registos. Diferentes métodos têm diferentes tempos de processamento e quantidades de informação transferida, juntamente com outro tipo de informação (como por exemplo o endereço), pode eventualmente criar-se uma espécie de padrão que permita facilitar o processo de deteção de anomalias;

- **cs-uri-stem**: Endereço relativo do pedido recebido. Neste campo, o endereço apenas se apresenta numa forma relativa que não inclui o nome de DNS ou o domínio. Este campo é depois processado e dividido entre outros campos denominados: *system*, *language*, *year*, *module*, *form* e *method*;

- **cs-uri-query**: Parte do endereço que contém os parâmetros. Embora teoricamente, o conteúdo destes parâmetros possam influenciar o desempenho, o seu impacto não é relativamente significativo, uma vez que o fator mais importante para qualquer análise de desempenho é a complexidade dos algoritmos usados;

- **port**: Porto onde o pedido foi recebido. Devido à configuração do servidor web, são usados três portos: 80, 443 e 4444. Como indicado anteriormente, cada site está configurado para responder a pedidos de um porto destes, assim sendo, os campos *port* e *s-sitename* são redundantes;

- **cs-username**: Nome de utilizador registado que executou o pedido. Por vezes este campo encontra-se vazio, uma vez que certos pedidos podem ser executados por utilizadores não autenticados no sistema (por exemplo, pedir a página inicial, ou a página de login). Este campo não contém informação útil para a monitorização numa forma geral, portanto, não será incluído na estrutura final.

- **c-ip**: IP de origem do pedido. Ainda que este campo não forneça nenhuma informação em termos de desempenho, poderá ser usada para encontrar padrões relativos a um certo IP. Esta informação pode ser útil para encontrar origens de ataques ou outro tipo de problemas. Este campo será incluído na estrutura final.

- **cs-version**: Versão de HTTP usada no pedido. Em todos os registos analisados, a versão é sempre HTTP/1.1, isto acontece visto que esta é a versão HTTP usada por todos os navegadores mais usados. Como o valor deste campo é basicamente redundante, não será incluído na estrutura final.

- **cs(User-Agent)**: User-agent do navegador do cliente que efetuou o pedido. Ainda que este campo possa conter informação útil para outros casos, não é especialmente útil para a monitorização, desse modo, não será incluído na estrutura final.

- **cs(Cookie)**: Cookies enviadas pelo cliente no pedido. Tal como o campo anterior, este campo poderá ser útil para outros casos mas não para a monitorização do sistema, e portanto, não será incluído na estrutura final.

- **cs(Referer)**: Endereços de uma página anterior que levou o cliente à página atual. Mais uma vez este campo também não contém informação útil para a monitorização e não será incluído na estrutura final.

- **cs-host**: URL do servidor que respondeu ao pedido, este pode ser o valor por omissão "erp.unl.pt"que indica o servidor de distribuição de carga, ou "erp1.unl.pt"ou "erp2.unl.pt"para os servidores da aplicação. Por norma, todos os pedidos devem ser direcionados para o servidor de distribuição de carga, no entanto, por vezes, os clientes podem, por algum motivo estar a executar pedidos específicos a um servidor específico que, por opção de desenho do sistema, podem não estar preparados para lhes responder. De modo a detetar erros deste tipo, este campo deverá ser mantido na estrutura final, no entanto, pode possivelmente ser guardado num formato diferente, por exemplo, um pequeno inteiro, 0, caso o servidor alvo seja o servidor de distribuição de carga, ou 1 e 2 para representar respetivamente os outros dois servidores;

- **sc-status**: Código HTML de estado do pedido. O valor deste campo será mantido na estrutura final pois pode ser usado para classificar pedidos em termos de normalidade. Um valor de 500 para o estado do pedido significa que ocorreu um erro interno no servidor, ou um valor de 503 significa que o serviço não se encontra disponível, possivelmente por carga excessiva, enquanto que valores do tipo 2XX significam que o pedido foi bem sucedido;

- **sc-substatus**: O sub estado do pedido. Este sub estado é um valor relativo ao estado guardado no campo *sc-status*. A Microsoft fornece uma lista com os diferentes valores possíveis de estado[1]. A informação presente neste campo não tem grande influência na monitorização, portanto este também não será incluído na estrutura final.

- **sc-win32-status**: O código de estado do processo do servidor. Estes códigos fornecem informações adicionais sobre que tipo de erros durante o processamento do pedido. Informação adicional sobre o significado de cada um destes códigos pode ser consultada na página oficial da Microsoft[2]. Este campo também não será incluído na estrutura final.

[1] https://support.microsoft.com/en-us/help/943891/the-http-status-code-in-iis-7-0-iis-7-5-and-iis-8-0
[2] https://docs.microsoft.com/en-us/windows/desktop/Debug/system-error-codes

- **sc-bytes**: Quantidade de informação enviada pelo servidor, em bytes. Pedidos semelhantes devem responder com informação semelhante, caso este valor varie muito da normalidade, pode indicar um problema nesse tipo de pedidos;

- **cs-bytes**: Quantidade de informação recebida pelo servidor, em bytes. Pode não haver uma relação direta entre o valor deste campo e um eventual problema no sistema, no entanto, este campo pode ter relevância para detetar possíveis ataques de negação de serviço (DOS).

- **time-taken**: Tempo, em milissegundos, durante o qual o pedido esteve a executar. Esta é a métrica mais importante a ser analisada, valores excessivamente altos neste campo indicam que existe algo a atrasar o sistema;

3.2.1.2 Estrutura de Dados para Processamento

De modo a minimizar a quantidade de informação a armazenar, foi efetuado uma análise dos campos apresentados na subsecção anterior. Depois dessa análise foram escolhidos aqueles campos com maior relevância para a monitorização. De seguida serão listados esses campos, será especificada que tipo de informação cada campo guarda e ainda, qual a sua utilidade. Será ainda indicado, de uma forma simples, como os valores destes campos são obtidos.

- **timestamp**: Este campo guarda um valor de tempo no formato timestamp, derivado dos campos originais *date* e *time*. Este campo pode ser usado para identificar a altura em que o evento foi realizado. Algumas utilizações para este campo incluem filtros ou agregações temporais.

- **client_ip**: Este campo guarda o endereço de ip do cliente que efetuou o pedido. Os valores guardados neste campo serão uma cópia direta do campo *c-ip* do ficheiro de registos e poderão ser usados para ajudar a encontrar possíveis problemas relativos a uma localização em específico.

- **client_name**: Este campo guarda o nome de cliente que está autenticado na aplicação web e que realizou o pedido registado. Os valores guardados neste campo serão essencialmente uma cópia do valor do campo *cs-username*, no entanto, será guardado um valor "None"caso o valor deste campo no ficheiro de registo apresente um valor -"(significando que não foi registado o nome de utilizador, o que acontece, por exemplo, em casos em que o utilizador ainda não efetuou o log in). Este campo pode ser útil para encontrar problemas relacionados com uma conta de utilizador em específico.

- **server_name**: Este campo guarda o nome do servidor que processou o pedido registado. O valor deste campo irá também ser apenas uma cópia do valor original, neste caso do campo *s-computername*. Com este campo podem ser implementados

filtros de modo a poder analisar o funcionamento de cada máquina, bem como o funcionamento geral do sistema.

- **status**: Este campo guarda o valor de estado do pedido após a sua execução. O valor guardado neste campo é uma cópia do valor do campo original *sc-status*. Com os valores deste campo, facilmente se conseguem encontrar alguns problemas como, por exemplo, erros de processamento, que resultam num valor de 500.

- **time_taken**: Este campo guarda informação sobre quanto tempo o pedido demorou a executar. Este é dos campos mais importantes da estrutura de dados. Quando comparando o seu valor a valores registados anteriormente no mesmo campo, é possível facilmente perceber se existem atrasos no sistema e ainda, quão graves são esses atrasos. O valor deste campo é obtido transformando o valor do campo original *time-taken* num valor inteiro.

- **sent**: Este campo guarda informação sobre a quantidade de informação enviada do servidor para o cliente. O valor para este campo é também obtido fazendo o parse do valor original de *sc-bytes* para um valor inteiro. Embora não seja algo tão significativo como o valor do tempo de processamento, o valor deste campo pode possivelmente indicar anomalias em pedidos que normalmente enviam a mesma quantidade de informação.

- **received**: Este campo guarda informação sobre a quantidade de informação recebida no servidor. Mais uma vez, o valor deste campo é obtido fazendo um parsing para inteiro do campo original *cs-bytes*. Tal como o campo *sent*, mesmo não sendo muito significativo, pode ter alguma utilidade. Quando combinados, os valores destes campos indicam a quantidade de informação transferida, com esta informação, podem ser eventualmente encontrados problemas com a rede.

- **method**: Este campo guarda que método HTTP foi usado no pedido. Esta informação é importante para analisar outras métricas, uma vez que, mesmo usando o mesmo endereço, métodos diferentes, podem executar algoritmos diferentes e portanto, poderão ter valores de tempo de processamento e quantidade de informação transferida diferentes. O valor deste campo é obtido através de uma cópia direta do valor do campo original *cs-method*.

- **url_target**: Este campo guarda o endereço URL indicado no pedido registado. Um dos fatores que mais influenciam as métricas de transferência de informação e tempo de processamento, é o URL que é invocado. Cada URL está relacionado com uma ação em específico, desse modo, para resolver o pedido, a aplicação poderá implementar algoritmos bastante distintos dos implementados para outros URLs. O valor deste campo é uma cópia direta do valor do campo *cs-uri-stem*.

Os campos listados acima são os principais campos da estrutura final e são todos obtidos com pouco ou nenhum processamento, relativamente aos correspondentes campos nos ficheiros de registos de eventos. Adicionalmente, a estrutura final inclui alguns outros campos derivados do campo *url-target*. Ainda que a informação presente nestes novos campos seja redundante, irá facilitar muito a construção de pedidos ao sistema de armazenamento e ainda, possibilitar a criação de certos tipos de filtros. Uma vez que é sabido à partida que maior parte dos endereços URL válidos irão seguir uma certa estrutura, é possível facilmente dividir os endereços pelos diferentes campos dessa estrutura, e assim, consegue-se obter ainda mais informação acerca do pedido. Depois de uma análise a várias entradas no ficheiro de registos, verificou-se que as seguintes estruturas são as mais usadas.

Listagem 3.2: Exemplo de algumas estrutura de endereços dos pedidos registados

```
1 # Estrutura principal completa com todos os campos
2 /<system>/<language>/<year>/<module>/<form>/<target>
3
4 # Estrutura sem o campo "module"
5 /<system>/<language>/<year>/<form>/<target>
6
7 # Estrutura pagina principal de sistema
8 /<system>
```

Certamente que estas não serão as únicas estruturas de endereços usadas. Desse modo, este projeto é realizado tendo em mente a necessidade de configurar outras estruturas. Posteriormente será apresentado em detalhe, o algoritmo de validação bem como, instruções para configurar novas estruturas com os seus respetivos campos. Com a análise realizada ao funcionamento da plataforma, foi possível perceber que, no máximo, a estrutura usada em pedidos para endereços válidos, inclui os seis campos indicados na estrutura principal na figura 3.2. Os significados de cada um desses seis campos serão descritos de seguida.

- **url_system**: Nesta solução em específico, a aplicação web serve vários sistemas para diferentes áreas de negócio. Os diferentes sistemas que estão a ser usados atualmente são os seguintes: GFP para gestão financeira, CIB para gestão patrimonial, GPT para gestão de projetos e GIP para gestão de recursos humanos.
- **url_language**: Este campo define a linguagem a usar na página pedida pelo cliente. Em todos os casos analisados durante este projeto, a linguagem usada é português.
- **url_year**: Este campo representa o ano de funcionamento sobre o qual se pretende obter a informação pedida.
- **url_module**: Este campo indica informação referente à estrutura em que a aplicação está implementada. O seu significado concreto não têm grande relevância para o estudo realizado neste documento, no entanto, uma vez que o valor deste campo

representa uma pequena parte do sistema, pode ser usado como filtro para os registos de eventos, de modo a tentar encontrar problemas relativos apenas a uma parte específica do sistema.

- **url_form**: Mais uma vez este campo refere-se a uma parte da estrutura interna da aplicação, podendo também ser usado de modo a filtrar registos e encontrar problemas relacionados com uma destas partes do sistema em específico.
- **url_action**: Este campo indica a função específica que foi chamada no pedido registado. Este será provavelmente um dos campos mais importantes desta estrutura visto que, filtrar os registos pelo valor deste campo, permite saber se existe ou existiu algum problema com alguma função em específico.

Para além dos campos indicados, é ainda adicionado um novo campo *url_target* que representa a página específica que foi invocada no pedido. Esta informação é obtida a partir do último campo do endereço. Outros endereços válidos que os clientes poderão invocar, incluirão um subconjunto dos campos da estrutura principal, como é o caso das estruturas sem o campo *module* e o endereço de página inicial de sistema, ambas as estruturas estão representadas na figura 3.2. De modo a indicar que estrutura foi usada para processar o endereço de cada pedido, será ainda acrescentado um outro campo *url_type*.

3.2.1.3 Configuração da Solução de Processamento de Registos

Relativamente à solução de processamento de registos, toda a informação analisada anteriormente é usada para implementar o ponto de entrada para iniciar o processamento, o script *main.py*. Este script é responsável por iniciar todo o processo de processamento de ficheiro de registos e de armazenamento da informação obtida.

Depois de uma análise dos ficheiros de registos, foram escolhidos os campos mais interessantes e úteis para efetuar a monitorização do sistema. Esses campos são configurados numa variável *required_fields*, caso futuramente seja necessário adicionar ou remover campos, basta alterar esta variável e o script adapta-se automaticamente.

Listagem 3.3: Variável de configuração dos campos necessários para cada registo

```
1 required_fields = [
2     "date", "time", "cs-uri-stem", "cs-username", "s-computername", "c-ip",
3     "cs-method", "sc-status", "sc-bytes", "cs-bytes", "time-taken"
4 ]
```

Para configurar o pré-processamento de registos é necessário definir uma função que irá receber como parâmetro, um objeto com o registo processado, incluindo apenas os campos necessários definidos em *required_fields*, e quer irá devolver um outro objeto do registo com a estrutura final que irá ser guardado posteriormente pelo sistema de armazenamento de dados escolhido.

Tal como referido na secção anterior, irá ser necessário executar um processamento adicional nos valores do campo do endereço invocado. Este processamento baseia-se em regras definidas usando expressões regulares que são definidas numa variável *url_rules*.

Listagem 3.4: Variável de configuração de padrões de endereços

```
url_rules = [
    {
        "regex" : "/(CIB|GIP|GFP|GPT)/(pt-PT)/(\d{4})/(\w+)/(\w+)/(\w+)",
        "fields": ["url_system", "url_language", "url_year", "url_module",
                   "url_form", "url_action"],
        "type"  : "FULL_URL"
    },
    {
        "regex" : "/(CIB|GIP|GFP|GPT)/(pt-PT)/(\d{4})/(\w+)/(\w+)",
        "fields": ["url_system", "url_language", "url_year", "url_form",
                   "url_action"],
        "type"  : "NO_MODULE_URL"
    },
    {
        "regex" : "/(CIB|GIP|GFP|GPT)",
        "fields": ["url_system"],
        "type"  : "SYSTEM_URL"
    },
    ...     # Outras regras de processamento de endereços
    {
        "regex" : "(.*)",
        "fields": ["url_action"],
        "type"  : "UNKNOWN"
    }
]
```

A variável *url_rules* guarda uma lista de objetos usados para definir os diferentes tipos de endereços. Cada objeto destes inclui um campo *regex* onde é definida a expressão regular a usar no processamento, onde são definidos os diferentes campos que se pretendem extrair do endereço. O campo *fields* define os nomes dos campos que serão gerados no objeto final do registo. Cada valor extraído, usando os grupos definidos na expressão regular, irá ser guardado num destes campos em específico pela ordem da definição dos nomes dos campos, ou seja, o valor obtido a partir do primeiro grupo será guardado num campo cujo nome é definido pelo primeiro valor da lista em *fields*, o segundo valor será guardado no campo definido pelo segundo valor da lista e assim sucessivamente para os restantes campos. Por fim, existe um outro campo *type* onde se pode definir um nome ou título para cada uma das estruturas de endereços que será também guardado na estrutura final do registo.

Com as duas variáveis definidas anteriormente já é possível definir a função de pré-processamento de registos. Esta função terá de ser enviada para a classe onde será realizado o processamento do ficheiro. O seguinte excerto de código representa a definição

desta função, bem como uma versão simplificada da mesma que será posteriormente detalhada.

Listagem 3.5: Função de Pré-processamento - Esqueleto

```
def preprocess_function(raw):
    log = {}

    # Process simple information
    ...

    # Process url
    ...

    # Find url match
    ...

    # Parse url
    ...

    return log
```

Tal como o excerto de código anterior indica, esta função de processamento requer um valor passado pelo argumento *raw* que representa cada um dos registos de cada ficheiro processado, num formato estruturado com os campos definidos em *required_fields*. Seguidamente irá ser criado um objeto *log* que será modificado ao longo do processamento realizado nesta função e que será devolvido como valor de retorno.

O primeiro passo ao implementar esta função é processar toda a informação mais básica, aquela informação que requer pouco ou nenhum pré-processamento. O seguinte excerto de código demonstra como isto foi implementado na solução desenvolvida neste projeto.

Listagem 3.6: Função de Pré-processamento - Processamento de Informação Simples

```
    ...
    # Process simple information
        datetime_str = "{} {}".format(raw["date"], raw["time"])
        log["timestamp"] = datetime.strptime(datetime_str, '%Y-%m-%d %H:%M:%S')

        log["client_ip"] = raw["c-ip"]
        client_name = raw["cs-username"]
        log["client_name"] = None if client_name == "-" else client_name

        log["server_name"] = raw["s-computername"]
        log["status"] = raw["sc-status"]
        log["time_taken"] = int(raw["time-taken"])
        log["sent"] = int(raw["sc-bytes"])
        log["received"] = int(raw["cs-bytes"])
        log["method"] = raw["cs-method"]
    ...
```

No excerto de código anterior, apenas o valor de *timestamp* requer algum uso de funções externas de outras bibliotecas para processar os valores de data e tempo, presentes do registo original, e criar um objeto de tempo válido.

Seguidamente irá ser demonstrado como foi implementado o processamento dos endereços usando as regras definidas anteriormente. Para realizar este processamento, primeiro será necessário determinar qual das regras definidas deve ser usada para processar o endereço do registo corrente.

Listagem 3.7: Função de Pré-processamento - Procura de Padrões em Endereços

```
1    ...
2    # Find url match
3    url_str = str.lower(raw["cs-uri-stem"])
4    matches = None
5    found = None
6    for rule in url_rules:
7        p = re.compile(rule["regex"], re.IGNORECASE)
8        matches = p.search(url_str)
9        if matches is not None:
10           found = rule
11           break
12   ...
```

Inicialmente o processamento começa com duas variáveis vazias: a variável *matches* e a variável *found*, que irão armazenar respetivamente, os valores dos grupos processados usando as expressões regulares e o objeto com o objeto de *url_rules* usado para processar o endereço. Seguidamente foi implementado um ciclo que itera todas as regras, compila cada uma das expressões regulares e usa-as para procurar a informação pretendida no endereço fornecido. Caso o endereço obedeça à regra testada, os valores dos diferentes irão ser guardados, pela ordem que se encontram no endereço, na variável *matches*, caso contrário, o valor da variável irá continuar vazio. Por fim, verifica-se se o processamento da expressão regular foi bem sucedido, verificando se o valor da variável *matches* já não se encontra vazio. Caso existam grupos guardados, a regra que estaria a ser processada nesse momento é guardada na variável *found* e o ciclo é terminado. De modo a assegurar o bom funcionamento do algoritmo, é importante que esteja sempre definida, no fim da lista de regras (*url_rules*), uma regra geral que seja sempre válida para qualquer endereço.

Depois de encontrar a regra para o processamento do endereço, bem como os grupos resultantes da aplicação dessa regra, o próximo passo é preencher os restantes campos do objeto de registo que será devolvido pela função de pré-processamento.

Listagem 3.8: Função de Pré-processamento - Processamento de Endereços

```
1    ...
2    log["url_type"] = found["type"]
3    fields = found["fields"]
4    nr_fields = len(fields)
5    if (nr_fields > 0):
```

```
6           for fid in range(0, nr_fields):
7               log[fields[fid]] = matches.group(fid + 1)
8           log["url_target"] = matches.group(nr_fields)
9
10      return log
```

Neste excerto de código começa-se por guardar o nome atribuído à regra usada para processar o endereço, que estaria guardada no campo *type* do objeto de regra usado. De seguida, define-se uma variável auxiliar, onde serão guardados os nomes dos campos para guardar os valores extraídos do endereço. Caso existam campos para guardar (ou seja, o número de campos é maior que zero), é executado um ciclo e, por cada um destes campos, é criado um campo com o mesmo nome no objeto *log* cujo valor corresponde ao valor guardado na mesma posição no objeto *matches*. Para obter esse valor no objeto *matches*, usa-se a função *group* com um argumento que define o índice do grupo que se pretende obter, neste caso usa-se o mesmo índice que foi usado para indexar a lista *fields*, no entanto, é necessario adicionar uma unidade, visto que o primeiro grupo guardado no objeto *matches* representa o endereço inteiro.

Neste momento, depois de implementado o código anterior, já está completa a função de pré-processamento, sendo que, o próximo passo é enviar a informação necessária para a classe que irá efetuar o processamento dos ficheiros.

Listagem 3.9: Inicialização do processamento de ficheiros de registos do servidor IIS

```
1  storage = ElasticStorage(
2      host="192.168.160.128",
3      index="monitoring_iis",
4      type="monitoring_log_iis"
5  )
6  parser = IISLogParser(storage=storage)
7  parser.set_required_fields(required_fields)
8  parser.set_preprocess_function(preprocess_function)
9  parser.parse_files_in_directory("../../Logs/IIS/erp1/W3SVC1")
10 parser.parse_files_in_directory("../../Logs/IIS/erp2/W3SVC1")
```

O código apresentado começa por inicializar um objeto da classe de armazenamento. Como neste caso, o sistema pretendido pare este efeito é o Elasticsearch, irá ser usada a classe *ElasticStorage* para efetuar a gestão de dados. Para inicializar esta classe podem ser configurados vários parâmetros, neste caso, esses parâmetros são: *host* para definir a localização da máquina, *index* para definir o nome do índice onde a informação será armazenada, e *type* para definir o nome do tipo de objeto a armazenar.

O objeto devolvido após a inicialização desta classe é depois passado como parâmetro para a classe de processamento *IISLogParser*. Esta nova classe é responsável por configurar e inicializar o processamento de ficheiros de registos do servidor IIS. Com o objeto resultante da inicialização desta classe, é possível definir os campos necessários usando a função *set_required_fields* e a função de pré-processamento usando a função

set_preprocess_function. Finalmente, para inicializar o processamento dos ficheiros de registo, é chamada a função *parse_files_in_directory*, passando a diretoria dos ficheiros como argumento. Ao chamar esta função numa destas diretorias, o do código irá automaticamente processar todos os ficheiros que não tenham sido processados anteriormente e guardar a informação processada no sistema de armazenamento indicado.

3.2.2 Registos da Aplicação GENIO

A aplicação GENIO é uma aplicação gerada pela plataforma de geração de código GENIO que implementa todas as funcionalidades do sistema integrado de gestão da Universidade Nova de Lisboa. Esta aplicação está configurada para também criar ficheiros onde regista certos eventuais erros que possam ocorrer durante o processamento de algum pedido. Estes registos são muito menos complexos que os registos do servidor IIS, no entanto podem fornecer informação adicional sobre problemas no código.

Durante o período de desenvolvimento deste projeto foi possível ter acesso a alguns exemplos de ficheiros de registo de erros desta aplicação, no entanto, não foi possível obter registos do mesmo intervalo de tempo dos registos analisados do servidor IIS. Este fator limita um pouco o desenvolvimento do projeto pois não será possível observar resultados concretos, ainda assim, foi feito um estudo sobre a estrutura destes ficheiros e serão apresentadas algumas possíveis formas de relacionar os registos dos dois sistemas.

Esta aplicação gera diferentes tipos de registos, para a análise realizada neste projeto foram, escolhidos os ficheiros que registam erros ocorridos na aplicação, visto que, estes são os mais informativos acerca de potenciais problemas ocorridos na aplicação. Tal como no caso dos ficheiros de registos do servidor IIS, os ficheiros de registos gerados pela aplicação GENIO guardam um registo por cada linha, no entanto, neste caso, a estrutura desses registos em cada uma dessas linhas é definida previamente e não é alterada ao longo de cada ficheiro nem de ficheiro para ficheiro.

Ao contrário dos registos do servidor IIS, estes ficheiros não incluem qualquer título para os campos presentes em cada registo, ainda assim, para este caso específico, é relativamente simples de inferir o seu significado, a partir dos valores registados. Seguidamente irá ser apresentado um exemplo destes registos.

Listagem 3.10: Exemplo de registo gerado pela aplicação GENIO

```
1 2018-01-05 00:06:39,611 [WebAPI_ProcessMessage] ERROR - GetAllMessages failed..
```

Ao analisar este registo, facilmente se identificam os campos de data e tempo, com os respetivos valores *2018-01-05* e *00:06:39,611* separados por um espaço. Seguidamente, entre parêntesis retos, está registado o contexto da mensagem, em certos casos pode representar o nome de utilizador que executa as funções, sendo que poderá incluir também o método invocado. Novamente separado por um espaço, encontra-se o valor *ERROR*, este valor indica que este registo se refere a um erro que ocorreu na aplicação. Outros valores possíveis para este campo são, por exemplo, *DEBUG* caso o registo represente um

registo especial de debug que foi registado em alguma situação específica que tenha sido configurada anteriormente no próprio código. Este campo poderá ser designado como o tipo de registo. Por fim, separado por espaços e um hífen, o resto da linha do registo inclui informação acerca do mesmo, num formato textual sem estrutura específica, assim, este campo poderá ser identificado como campo de informação.

Visto que a estrutura é sempre a mesma para todos os ficheiros, para iniciar o processamento deste tipo de registos não será necessária configuração adicional. Caso no futuro se pretenda executar algum pré-processamento, será possível fazê-lo, sendo para isso necessário definir a respetiva função, com as regras de pré-processamento necessárias, e de seguida, enviá-la para a classe de processamento, tal como foi implementado para o caso dos registos do servidor IIS.

Listagem 3.11: Inicialização do processamento de ficheiros de registos da aplicação GENIO

```
storage = ElasticStorage(
    host="192.168.188.128",
    index="monitoring_genio",
    type="monitoring_log_genio"
)
parser = GenioLogParser(storage)
parser.parse_files_in_directory("../../Logs/Errlog/files")
```

3.3 Processamento de Registos de Eventos

Neste capítulo irá ser apresentado todo o trabalho realizado sobre o processamento de registos de eventos. Este processamento é dividido em várias partes. Primeiro foi implementada uma classe principal *LogParser* para gerir os ficheiros já processados, bem como abrir os ficheiros não processados para começar o seu processamento. Outras classes, como por exemplo *IISLogParser* e *GenioLogParser*, são mais específicas a tipos de registos de sistemas em concreto. Estas classes foram criadas para implementar a parte do processamento específico aos ficheiros desses sistemas.

3.3.1 Processamento de Ficheiros Usando a Classe *LogParser*

A classe LogParser é classe responsável por iniciar o processamento de ficheiros de registos de eventos de uma certa diretoria. Esta classe é responsável por abrir os ficheiros que se pretendam processar, separá-los em linhas e inicializar o processamento de cada registo de cada linha. Adicionalmente, esta classe gera ainda informação acerca de que ficheiros já foram processados, e quantas linhas foram processadas em cada ficheiro, de modo a que nenhum ficheiro seja processado mais do que uma vez. Durante a sua implementação, o principal foco foi em implementar todas as funcionalidades e partes do algoritmo que sejam transversais ao tipo dos registos que estejam guardados em ficheiros de texto.

Como já referido, esta classe implementa todas funcionalidades necessárias para executar o processamento de ficheiros de texto. Este fator, para além de facilitar a implementação posterior da lógica específica necessária para processar cada tipo específico de registos, possibilita ainda a fácil implementação de funcionalidades para permitir o processamento de registos provenientes de outras fontes. Ainda que não sejam demonstrados nenhum destes caso de uso neste projeto, esta funcionalidade poderá ser especialmente útil para casos onde os registos estejam guardados, por exemplo, em outras bases de dados.

A estrutura deste tipo de classes é bastante simples, como é comum a todas as classes, é necessário definir um construtor. O construtor para esta classe limita-se a receber e guardar localmente um objeto para a comunicação com o sistema de armazenamento que se pretende usar durante o processamento dos ficheiros. Para além do construtor, existem duas outras funções que terão de ser definidas para garantir o bom funcionamento da classe, estas funções são *parse_line* e *parse_files_in_directory*. De seguida será apresentado um excerto de código que mostra, de uma forma simplificada como esta classe foi implementada.

Listagem 3.12: Classe LogParser - Esqueleto

```
1 class LogParser:
2     def __init__(self, storage):
3         self.storage = storage
4
5     def parse_line(self, line):
6         pass
7
8     def parse_files_in_directory(self, directory):
9         ...
```

Neste excerto de código estão a ser definidas as funções indicadas anteriormente. A função *parse_line*, serve neste momento apenas como um placeholder e terá de ter uma implementação concreta em todas as subclasses desta classe. Esta função irá receber uma linha do ficheiro como argumento e terá de devolver um registo completamente processado (incluindo o processamento necessário definido nas funções de pré-processamento) e pronto para ser inserido na base de dados.

Quanto à função *parse_files_in_directory*, esta é a função principal da classe *LogParser*. A função recebe a diretoria dos ficheiros a processar, abre cada um desses ficheiros, itera as suas linhas, envia o conteúdo dessas linhas para a função *parse_lines*, invoca as funções necessárias para armazenar os registos obtidos e faz também a gestão da informação sobre que registos já foram processados.

Listagem 3.13: Classe LogParser - Ciclo principal

```
1 def parse_files_in_directory(self, directory):
2     log_files = os.listdir(directory)
3     for log_file in log_files:
```

```
4         path = os.path.abspath(directory + "/" + log_file)
5         ...
```

Esta função começa por obter uma lista de ficheiros presentes na diretoria indicada usando a função *os.listdir*. Esta função devolve uma lista de nomes de ficheiros. Seguidamente são iterados todos esses nomes de ficheiros presentes na lista, para cada um deles, irá ser obtida a localização absoluta desse ficheiro recorrendo à função *os.path.abspath*.

Listagem 3.14: Classe LogParser - Obtenção de informação sobre processamentos anteriores

```
1         ...
2         # Get previous parsing information
3         parse_info = self.storage.get_parse_info(path)
4         if parse_info and parse_info["finished"]:
5             print("File already parsed, skipping...")
6             continue
7         ...
```

O próximo passo é tentar perceber se este ficheiro já foi processado anteriormente. Para obter esta informação, é passado como argumento a localização absoluta do ficheiro, para a função *get_parse_info* da classe de armazenamento guardada na variável *storage*. Esta função pode devolver um objeto com a informação acerca do processamento executado anteriormente para aquele ficheiro, ou em alternativa, pode devolver um resultado nulo caso o ficheiro ainda não tenha sido processado. Esse resultado dessa função é guardado na variável *parse_info*, caso essa variável não esteja vazia e o valor presente no campo *finished* seja verdadeiro, este ficheiro é ignorado pois já foi processado anteriormente.

Com a informação acerca do processamento do ficheiro atualmente a ser processado, irá ser preparado o seu processamento, começando-se por definir algumas variáveis úteis de serem usadas posteriormente, ao longo do algoritmo.

Listagem 3.15: Classe LogParser - Abertura de ficheiro para leitura

```
1         ...
2         logs = []
3         nr_lines = 0
4         nr_lines_processed = 0
5         start_line_nr = parse_info["end_line_nr"] if parse_info else 0
6
7         try:
8             f = open(path, "r", encoding="ISO-8859-1")
9             lines = f.readlines()
10            nr_lines = len(lines)
11
12            ...
13
14        except Exception as e:
15            # Log the error to an external file
16
```

```
17              if f: f.close()
18              continue
19          ...
```

A variável *logs* irá guardar temporariamente em memória os registos antes de serem enviados para a classe de armazenamento e armazenados pelo sistema implementado nessa classe. A variável *nr_lines* guarda o número de linhas que o ficheiro atual contém, inicialmente o valor desta variável é zero mas será alterado posteriormente. A variável *nr_lines_processed* irá guardar o número de linhas processadas, inicialmente este valor é zero. A variável *start_line_nr* guarda o número da linha onde será inicializado o processamento, caso este ficheiro já tenha sido parcialmente processado, a linha inicial será linha seguinte à ultima processada anteriormente, caso contrário, a linha inicial será a primeira linha do ficheiro.

Após a inicialização das variáveis existe um bloco try/catch de modo a apanhar todas as possíveis exceções que possam ocorrer durante o processamento do ficheiro, desse modo, em vez de terminar o processamento por completo, o código poderá registar informação sobre o problema encontrado num ficheiro externo, continuando o processamento para os restantes ficheiros. Dentro do bloco principal, o código começa por abrir o ficheiro para leitura, usando a opção "r"e definindo a codificação do ficheiro como "ISO-8859-1"para evitar possíveis erros de leitura. Seguidamente é usada a função *readlines* para obter uma lista de linhas do ficheiro que serão guardadas numa variável *lines*, e ainda, é atualizado o valor da variável *nr_lines* com o número correto de linhas totais do ficheiro.

Ainda dentro do primeiro bloco de try/catch, é implementado um outro bloco deste tipo, permitindo assim que, para certos tipos de exceções ainda seja possível guardar alguma da informação temporária antes de parar o processamento do ficheiro atual.

Listagem 3.16: Classe LogParser - Processamento das linhas do ficheiro

```
1               ...
2               try:
3                   for line in lines:
4                       log = self.parse_line(line)
5                       nr_lines_processed += 1
6                       if nr_lines_processed >= start_line_nr:
7                           if log: logs.append(log)
8                           if nr_lines_processed % 100000 == 0:
9                               self.storage.save_logs(logs)
10                              logs = []
11
12                              end_line_nr = start_line_nr + nr_lines_processed
13                              info = {
14                                  "date_parsed": datetime.now(),
15                                  "path": path,
16                                  "nr_lines": nr_lines,
17                                  "start_line_nr": start_line_nr,
18                                  "end_line_nr": end_line_nr,
```

```
19                              "finished": False
20                          }
21                          self.storage.save_parse_info(info)
22                  except Exception as e:
23                      if len(logs) > 0:
24                          self.storage.save_logs(logs)
25                          end_line_nr = start_line_nr + nr_lines_processed
26                          info = {
27                              "date_parsed": datetime.now(),
28                              "path": path,
29                              "nr_lines": nr_lines,
30                              "start_line_nr": start_line_nr,
31                              "end_line_nr": end_line_nr,
32                              "finished": False
33                          }
34                          self.storage.save_parse_info(info)
35
36                      # Log the error to an external file
37
38                      f.close()
39                      continue
40              ...
```

Por cada linha presente na lista *lines*, é enviado o seu conteúdo para a função *parse_line* (que terá de ser implementada por alguma subclasse) e é obtido um objeto de registo que será armazenado pela classe *storage*. Em alternativa, o resultado dessa função poderá ser um valor vazio, o que significa que não foi possível obter um objeto de registo com a informação presente naquela linha, por exemplo, se a linha inclui apenas um comentário. Após o processamento, é incrementado o valor da variável que guarda o número de linhas processadas. Caso esse número seja igual ou superior ao número da linha a partir da qual se encontram as linhas não processadas, o registo processado será adicionado à lista temporária (caso não esteja vazio). Seguidamente, caso já tenham sido processadas um número suficientemente grande de linhas, a lista temporária será enviada para a classe de *storage* para serem armazenados os registos nela contidos, essa lista é depois limpa, e por fim, é ainda guardada a informação acerca das linhas processadas.

Por fim, após terminar o ciclo que itera as linhas do ficheiro, pode ainda ser necessário guardar alguma informação temporária, bem como, guardar a informação que indique que o ficheiro atual foi completamente processado com sucesso.

Listagem 3.17: Classe LogParser - Finalização do processamento

```
1       ...
2       # Save parsing information and remaining data
3       if nr_lines_processed > 0:
4           self.storage.save_logs(logs)
5           info = {
6               "date_parsed": datetime.now(),
7               "path": path,
```

```
                "nr_lines": nr_lines,
                "start_line_nr": start_line_nr,
                "end_line_nr": start_line_nr + nr_lines_processed,
                "finished": True
            }
            self.storage.save_parse_info(info)

            print("Progress: {}/{}".format(nr_lines_processed, nr_lines))
            print("Finished processing file:")
            print("Starting line:\t{}".format(start_line_nr))
            print("Nr processed:\t{}".format(nr_lines_processed))
        else:
            print("No lines processed")

        f.close()
```

Caso tenham sido processadas linhas durante o ciclo anterior, serão armazenados os restantes registos presentes na variável temporária *logs* e será ainda guardada informação acerca do estado do processamento deste ficheiro, no entanto, desta vez, esta informação é guardada com o campo *finished* com um valor verdadeiro de modo a garantir que o mesmo ficheiro não volta a ser processado. Por fim é libertado o ficheiro usando a função *close* do objeto de ficheiro.

3.3.2 Processamento de Ficheiros do Servidor IIS

Na secção anterior foi descrita a classe *LogParser*. Esta classe é uma das classes mais complexas neste projeto, no entanto, irá facilitar imenso o desenvolvimento de subclasses para processamento de ficheiros de sistemas específicos. Nesta secção irá ser descrita uma destas classes, a classe *IISLogParser*, responsável por processar os ficheiros de registos gerados pelo servidor web IIS. Visto que esta classe irá ser implementada como uma subclasse que estende a classe *LogParser*, só será necessário implementar funções para processar cada linha dos ficheiros, visto que toda a lógica de gerir ficheiros e informação do processamento, já estará a ser realizada na classe principal.

Listagem 3.18: Classe IISLogParser - Definição e inicialização

```
class IISLogParser(LogParser):
    def __init__(self, storage):
        self.req_fields = []
        self.preprocess = lambda x : x
        super().__init__(storage)
    ...
```

Para definir esta nova classe, é necessário passar a classe principal como argumento na definição desta. De seguida, é definido o construtor da classe. Este construtor terá de receber um objeto de classe de armazenamento, que terá de enviar para a classe principal. Ainda no construtor também serão definidas algumas variáveis de classe. A variável

req_fields guarda uma lista com os nomes dos campos que será necessário extrair dos ficheiros de registo. Esta variável é inicializada com uma lista vazia mas poderá ser configurada posteriormente. A variável *preprocess* guarda uma função de pré-processamento de registos que também poderá ser configurada posteriormente, sendo que, inicialmente, esta variável contém apenas uma função simples que devolve o argumento que recebe. Por fim, será invocado o construtor da classe principal com o argumento necessário, o objeto da classe de armazenamento.

Ainda que a única função estritamente necessária seja a função *parse_line*, foram criadas outras duas funções de modo a permitir a configuração do processamento de cada linha, que serão usadas em ficheiros de configuração como o *main.py* analisado em capítulos anteriores. Uma destas funções é a função *set_required_fields* que permite configurar que campos serão extraídos de cada registo do sistema IIS.

Listagem 3.19: Classe IISLogParser - Função *set_required_fields*

```
1    ...
2    def set_required_fields(self, required_fields):
3        self.field_ids = {}
4        for field in required_fields:
5            self.field_ids[field] = 0
6        self.req_fields = required_fields
7    ...
```

Neste excerto de código, para além de ser atualizada a lista de campos necessários, é ainda criada uma variável de classe chamada *field_ids*. Esta variável é usada para guardar a posição relativa de cada campo nas linhas que contem registos. Uma vez que inicialmente a informação da posição dos campos não é conhecida, este objeto de posições será ser inicializado com todas as posições a zero.

Listagem 3.20: Classe IISLogParser - Função *set_preprocess_function*

```
1    ...
2    def set_preprocess_function(self, preprocess_function):
3        self.preprocess = preprocess_function
4    ...
```

A função *set_preprocess_function* é bastante simples, apenas recebe uma função como argumento e guarda essa função na variável de classe *preprocess* para ser utilizada posteriormente durante o processamento dos registos.

Listagem 3.21: Classe IISLogParser - Função *parse_line*

```
1    ...
2    def parse_line(self, line):
3        words = line.strip().split("␣")
4
5        # Handle comments
6        if words[0][0] == "#":
7            if words[0] == "#Fields:":
```

```
8                   for w in range(0, len(words)):
9                       word = words[w]
10                      if word in self.field_ids:
11                          self.field_ids[word] = w - 1
12          # Handle log entries
13          else:
14              raw = {}
15              for field in self.req_fields:
16                  raw[field] = words[self.field_ids[field]]
17
18              log = self.preprocess(raw)
19              return log
```

A função *parse_line* é responsável por implementar o processamento de cada linha. Essa linha será recebida como uma string por argumento, serão removidos os caracteres de retorno usando a função *strip*, e separados por espaço, todos os valores dos diferentes campos, que serão guardados na variável *words*.

Depois de ter todos os campos ou palavras da linha separados, é necessário verificar se a linha contém um comentário ou um registo. Para determinar se a linha representa um comentário, basta verificar se o primeiro caractere da linha é um símbolo "#". Caso este seja o caso, é importante também verificar se este comentário contém o índice de campos. Neste tipo de comentário especial, para além da linha começar com o caractere "#", a linha começa com a palavra "#Fields:". Caso se verifique que se está a processar um comentário com o índice de campos, para cada campo nesse índice, é guardada, na variável *field_ids*, a sua posição relativa na linha.

Para os casos em que a linha representa um registo, inicialmente, começa-se por criar um objeto vazio, numa variável *raw*, para guardar os valores dos campos necessários do registo. De seguida, para cada campo necessário definido anteriormente, é obtida a sua posição relativa usando a variável *field_ids*. Com essa posição é possível obter o respetivo valor que será guardado no objeto *raw*. Depois de todos os campos preenchidos, o objeto raw é enviado para a função de processamento, sendo que o seu resultado será também depois devolvido pela função *parse_line*.

3.3.3 Processamento de Ficheiros da Aplicação GENIO

Os ficheiros de registos de eventos gerados pela aplicação GENIO são relativamente mais simples do que os ficheiros gerados pelo sistema IIS, o que resulta em menos opções de configuração possíveis de serem implementadas. Tal como a classe analisada na secção anterior, esta classe também estende a funcionalidade da classe *LogParser*, adicionando apenas lógica específica ao processamento dos ficheiros de registos da aplicação GENIO.

Listagem 3.22: Classe GenioLogParser - Definição e inicialização

```
1 class GenioLogParser(LogParser):
2     def __init__(self, storage):
3         self.preprocess = lambda x : x
```

```
4         super().__init__(storage)
5     ...
```

A definição da classe e a sua inicialização é muito semelhante ao já visto anteriormente. Neste caso, uma vez que os campos deste tipo de ficheiros são sempre os mesmos, na mesma estrutura, para todos os ficheiros a analisar, não será necessário definir uma variável para guardar os campos que se pretende analisar. Quanto ao resto do código, começa-se novamente por receber a classe principal na definição da classe de modo a definir que esta classe é uma subclasse de *LogParser*. A inicialização irá receber também um objeto com uma classe de armazenamento, inicializar a variável *preprocess* com uma função simples que devolve o argumento e por fim, inicializar a classe principal usando o argumento recebido.

Listagem 3.23: Classe GenioLogParser - Função *set_preprocess_function*

```
1     ...
2     def set_preprocess_function(self, preprocess_function):
3         self.preprocess = preprocess_function
4     ...
```

Relativamente a funções auxiliares só será implementada a função para definir a função de pré-processamento dos registos, exatamente igual à vista anteriormente na classe para processar os ficheiros IIS. Ainda que, durante a implementação deste projeto, não se tenha encontrado regras para implementar nesta função, ela foi implementada para facilitar o desenvolvimento deste tipo de regras no futuro.

Listagem 3.24: Classe GenioLogParser - Função *parse_line*

```
1     ...
2     def parse_line(self, line):
3         line_regex = "(.*?␣.*?)␣\[(.*?)\]␣(.+?)␣-␣(.+)"
4         p = re.compile(line_regex, re.IGNORECASE)
5         matches = p.search(line)
6
7         raw = {
8             "timestamp":
9                 datetime.strptime(matches.group(1), '%Y-%m-%d␣%H:%M:%S,%f'),
10            "context": matches.group(2),
11            "type": matches.group(3),
12            "info": matches.group(4)
13        }
14
15        log = self.preprocess(raw)
16        return log
17    ...
```

A função de processamento de linha usa um padrão definido uma expressão regular, de modo a extrair toda a informação da linha do registo. Este padrão foi implementado após a análise dos registos que foi apresentada no capítulo anterior. Neste capítulo

verificou-se que os registos eram compostos por 4 campos. Em primeiro lugar encontra-se uma representação de tempo de quando o evento foi registado. Essa representação de tempo irá depois ser guardada no campo *timestamp* do objeto de registo. De seguida, separado por espaço e entre parêntesis retos, encontra-se a informação do contexto deste evento. Este valor será guardado no campo *context*. De seguida, encontra-se o campo que indica o tipo do evento. Os valores deste campo serão guardados no campo *type*. Por último encontra-se informação textual acerca do evento que será guardada no campo *info*. Depois de completo, o objeto que representa o registo de evento será enviado para a função de pré-processamento. O resultado dessa função será depois devolvido pela função *parse_line*.

3.4 Implementação do Painel de Controlo

Para implementação do painel de controlo, o primeiro passo é realizar a instalação do sistema Kibana, para tal, basta descarregar o software, através da sua pagina oficial. Depois de instalado o sistema Elasticsearch, configurada corretamente a sua ligação com o Python, e definidas as estruturas de dados usadas, é possível começar a criar tipos de visualizações e painéis de controlo. Alguns destes tipos de visualizações mais relevantes, irão ser descritos de seguida.

3.4.1 Visualização da Linha Temporal

A linha temporal é o tipo de visualização mais importante neste tipo de painéis de controlo. De uma forma simples é possível analisar a evolução de várias métricas ao longo do tempo. No sistema Kibana, este tipo de visualização pode ser definido usando o tipo de visualização *TSVB*. Seguidamente terá de ser indicado o tipo de operação e o campo, que se pretende usar para a agregação de dados, neste caso, pretende-se que o gráfico mostre a evolução da soma dos valores de tempo de processamento.

Visto que os registos são gerados por dois servidores diferentes, será também interessante poder avaliar a evolução do desempenho de cada um deles. De modo a apresentar esta informação, no mesmo gráfico, serão apresentadas duas séries de dados, sendo que, para cada uma delas, será definido um filtro para filtrar o tipo de servidor.

A figura 3.4 demonstra o resultado final da definição deste tipo de visualização. No mesmo gráfico encontram-se duas séries de dados, com diferentes cores, relativas aos dois servidores erp1 e erp2, desta forma é fácil de os comparar e possivelmente encontrar problemas específicos a cada um deles.

3.4.2 Visualização de Tempo de Processamento

O objetivo desta visualização de tempo de processamento, é perceber, de uma forma rápida, qual a distribuição de pedidos por tempo que demoram a ser executados. Para cada intervalo de valores de tempo de processamento, será apresentado o número de

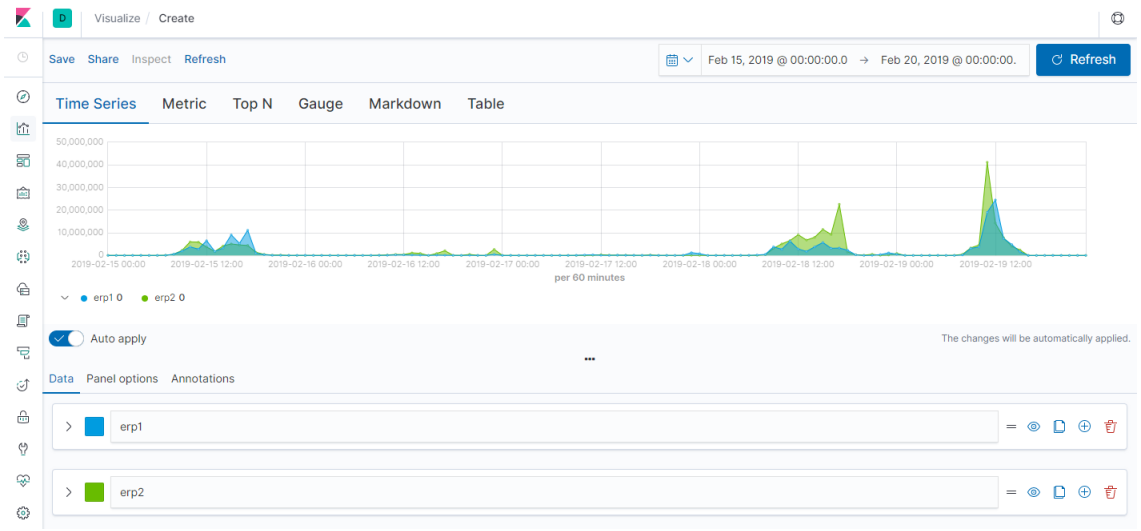


Figura 3.4: Implementação da Visualização da Linha Temporal

pedidos que nele se incluem. Adicionalmente, o Kibana permite que, com apenas um clique num destes grupos, toda a informação to painel de controlo, será filtrada para apenas mostrar informação acerca dos registos desse grupo. Esta funcionalidade é muito útil para, por exemplo, analisar facilmente apenas os registos com um elevado tempo de processamento.

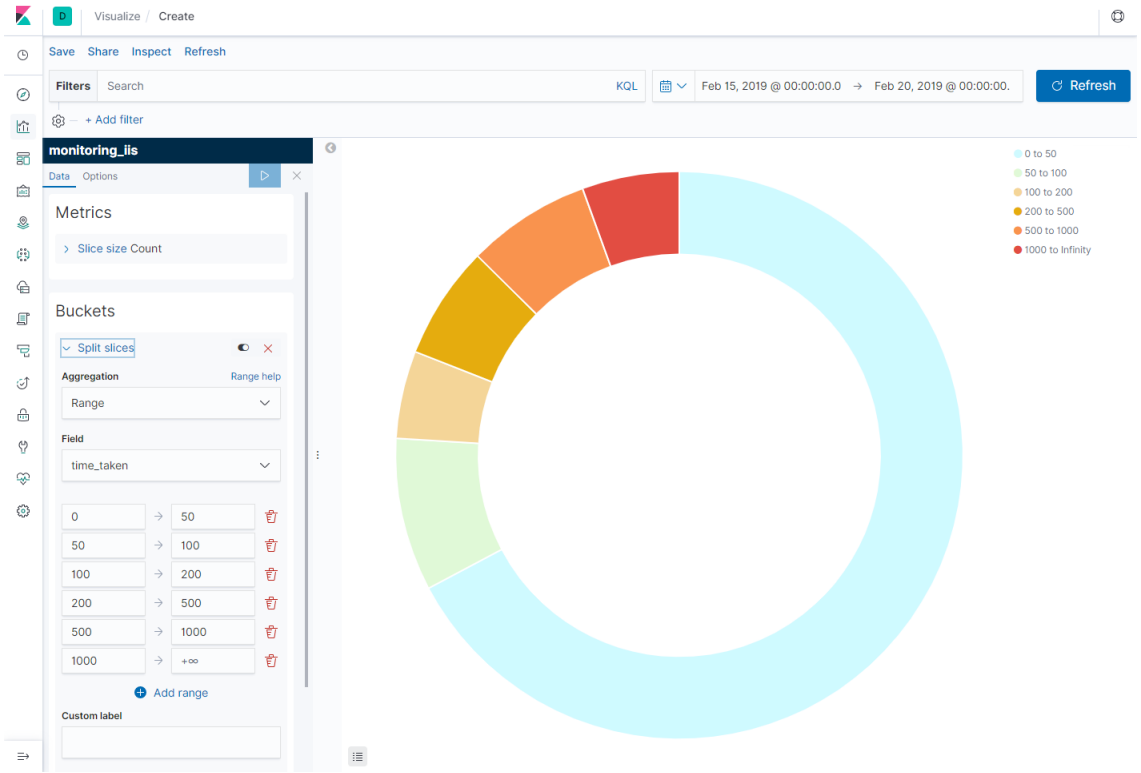


Figura 3.5: Implementação da Visualização de Tempo de Processamento

A figura 3.5 demonstra este tipo de visualização. Neste caso, foram definidos alguns dos grupos descritos anteriormente, e foi pedido ao sistema para calcular a soma dos elementos de cada grupo e apresentar essa informação, numa forma relativa, num gráfico tipo tarte. Adicionalmente, é possível personalizar o código de cores a usar nos diferentes grupos, sendo que neste caso, definiram-se cores mais vibrantes quando os grupos com maiores valores, de modo a facilitar a sua leitura.

3.4.3 Visualização de Tipo de Sistema

Embora este tipo de visualização seja bastante específico ao sistema a monitorizar, estudado durante este projeto de dissertação, é interessante de ser analisado pois pode facilmente ser adaptado para outros casos. A visualização do tipo de sistema permite analisar a distribuição dos pedidos pelos sistemas implementados dentro do portal de gestão analisado.

Para além de informação estatística, esta visualização aqui apresentada possui ainda outras utilizações. Para realizar a analise de cada sistema, o painel de controlo da solução anterior, implementava cópias de todos os tipo de visualização, cada uma delas relativa a cada sistema. Da forma apresentada nesta nova solução só será necessário definir cada tipo de visualização uma única vez, sendo que a informação relativa a cada sistema poderá ser facilmente filtrada, usando esta visualização de tipo de sistema, com apenas um clique no grupo do respetivo sistema.

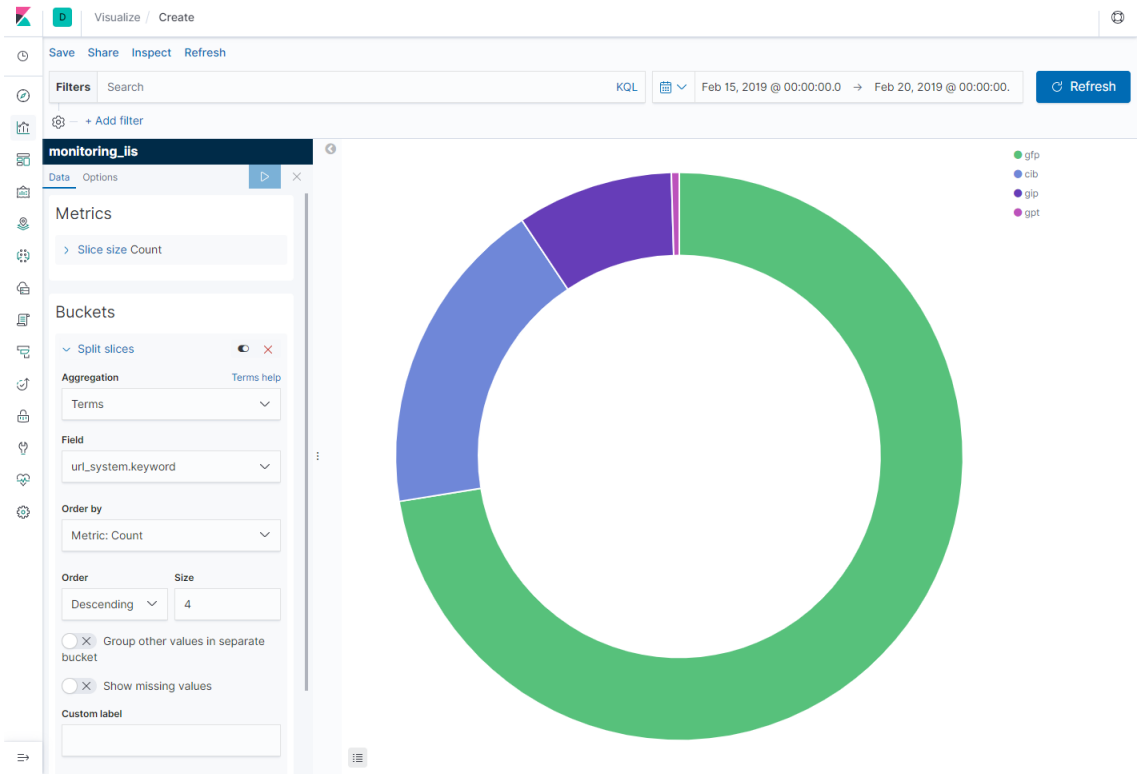


Figura 3.6: Implementação da Visualização de Tipo de Sistema

A figura 3.6 apresenta o resultado da implementação deste tipo de visualização. No gráfico de tarte gerado, estão representados os quatro sistemas disponíveis: gfp, cib, gip, gpt. Ao contrário da visualização de tempo de processamento, para estes casos, os grupos serão automaticamente criados pelo próprio Kibana. Ao definir o tipo de agregação como *Terms*, o campo a usar como *url_system* e o número de grupos que se pretendem obter, o sistema irá criar um grupo por cada um dos diferentes valores presentes esse campo.

3.4.4 Visualização de Tabela de Registos

A visualização de tabela de registos permite ao utilizador ler toda a informação armazenada acerca dos registos num formato tabular. Ainda que este tipo de visualização não seja particularmente útil quando o número de registos é demasiado elevado, torna-se útil para quando estão definidos filtros suficientes para realizar uma análise mais pormenorizada da informação.

Tal como para os outros tipos de visualização, também é possível definir filtros baseados na informação apresentada na tabela. Para estes tipos de visualização em tabela, é possível facilmente adicionar filtros para filtrar qualquer valor, de qualquer campo. Ao passar o cursor por cima de qualquer valor, o sistema Kibana apresenta opções para adicionar um filtro baseado nesse mesmo valor. Adicionalmente, é possível ainda alterar a ordem de apresentação dos registos, usando as opções para o efeito, presentes no título de cada um dos campos.

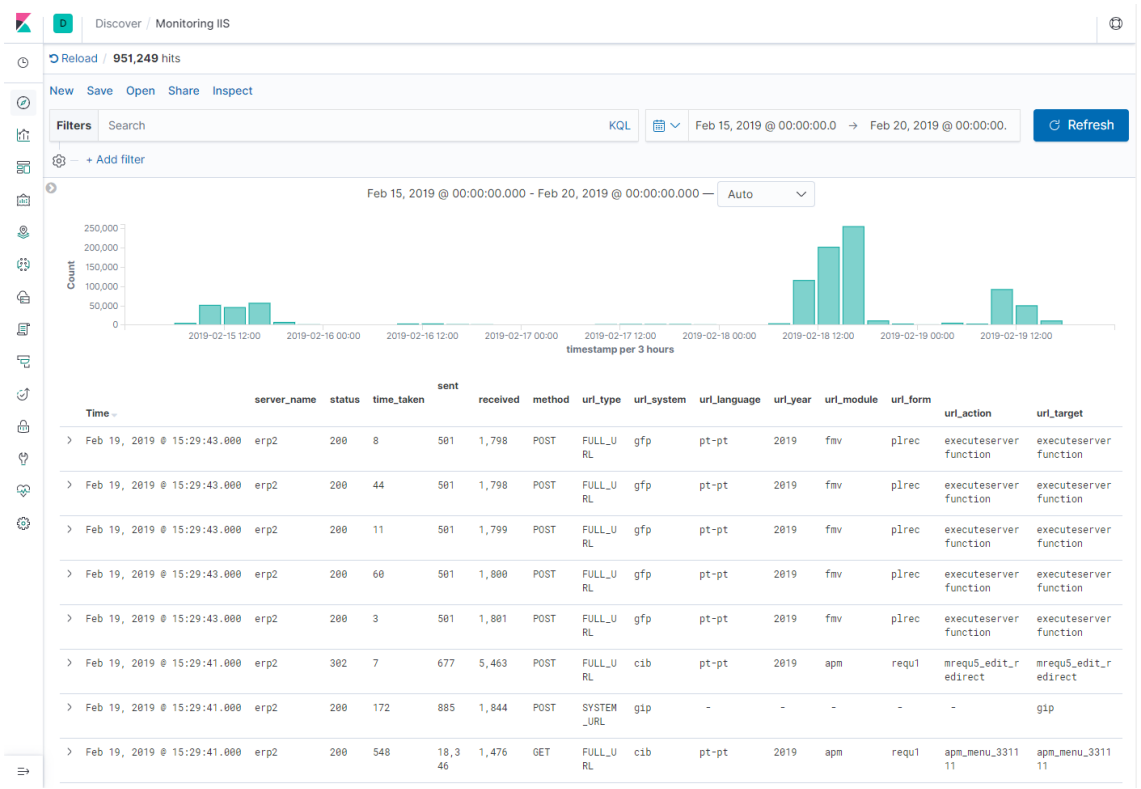


Figura 3.7: Implementação da Visualização de Tabela de Registos

Na figura 3.7 está representado o resultado final da implementação de uma destas tabelas. Supostamente, nesta visualização deveriam estar apresentados todos os campos presentes nos registos, no entanto, para efeitos de exemplo, foram removidos alguns destes campos por conterem informação confidencial.

3.4.5 Painel de Controlo

Depois de todas as visualizações criadas, é possível criar um painel de controlo onde estas serão dispostas da forma pretendida.

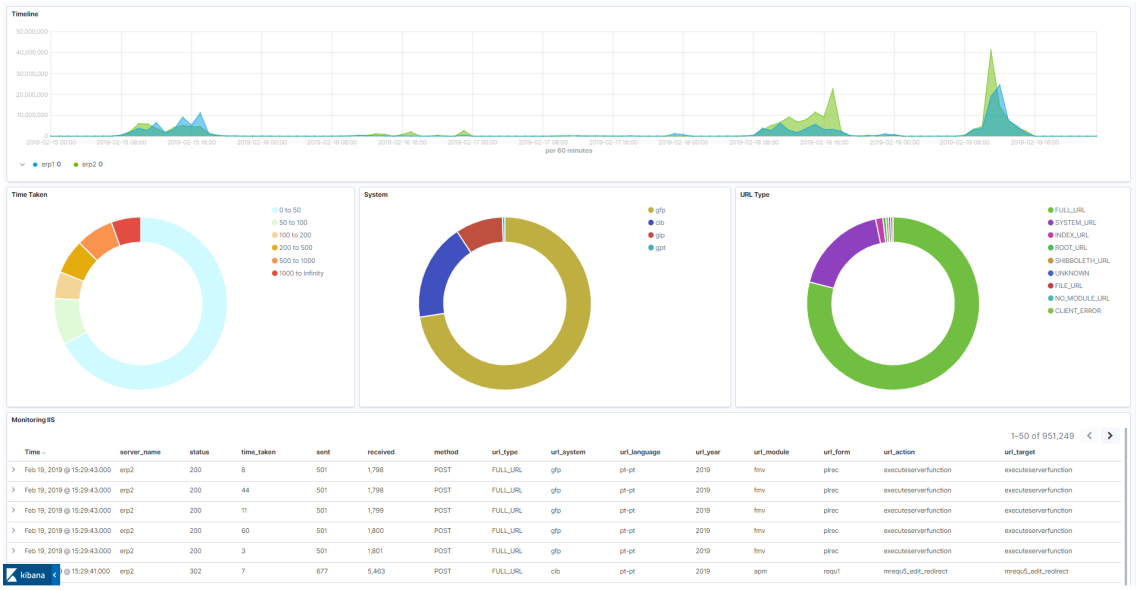


Figura 3.8: Implementação do Painel de Controlo

Na figura 3.8 está apresentado o resultado final da implementação do painel de controlo. A informação está disposta de uma forma específica, com objetivo de melhorar o mais possível a sua leitura. Em primeiro lugar encontra-se a linha temporal. Este gráfico permite rapidamente perceber em que alturas os servidores tiveram uma maior carga de processamento. De seguida, encontram-se os gráficos em tarte com outro tipo de informação estatística. Finalmente, uma vez que a informação presente na tabela não será muito útil até estarem definidos filtros suficientes, esta tabela aparece por baixo dos restantes elementos visuais.

Capítulo 4

Análise de Resultados e Conclusões

Neste capítulo irá ser resumido o trabalho realizado, analisados os resultados obtidos tendo em conta os objetivos iniciais e ainda, apresentadas algumas conclusões bem como propostas para trabalho futuro.

4.1 Resultados

O objetivo principal deste projeto foi melhorar a solução de monitorização da Quidgest. Esta solução tinha alguns problemas como informação mal processada guardada na base de dados de registos, demasiada informação armazenada e ainda, mau desempenho do painel de controlo.

De forma a melhorar o processamento da informação foram criados vários scripts Python que auxiliam esse processamento. Com a solução atual é possível definir exatamente a estrutura de dados final e de que forma processar os dados para os inserir nessa estrutura. Desta forma, não só se consegue processar corretamente toda a informação, como também se torna possível incluir, no futuro, outro tipo de ferramentas no processamento, como por exemplo, ferramentas de análise baseadas em técnicas de aprendizagem automática.

Quanto à quantidade de informação armazenada, ainda que não tenha sido possível medir a quantidade de informação armazenada pela solução antiga, foi realizado um teste no ambiente de utilização real que produziu muito bons resultados.

A figura 4.1 representa os resultados obtidos no teste de armazenamento realizado. Para este teste, usando a solução apresentada neste documento, foram processados aproximadamente 160 GB de ficheiros de registos de eventos do servidor IIS. Após finalizado o processamento dos registos, foi analisada a dimensão do índice Elasticsearch onde toda a informação foi armazenada. O espaço em disco ocupado pela informação guardada

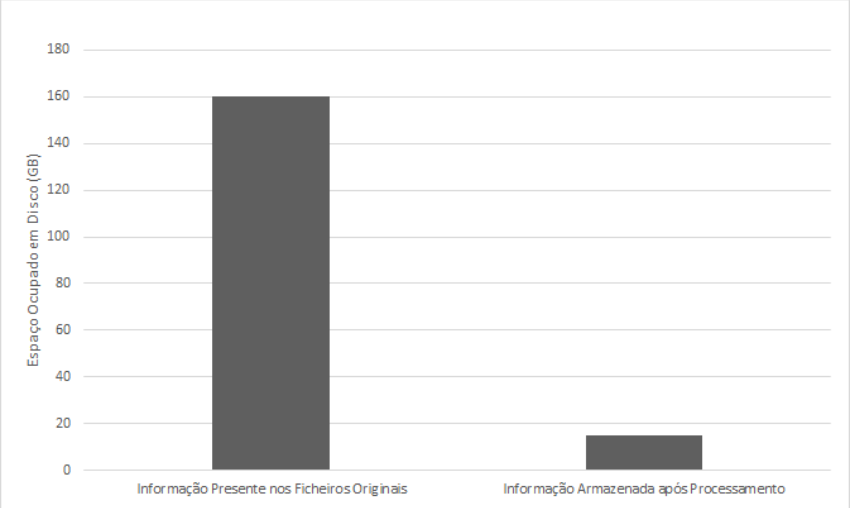


Figura 4.1: Resultados Finais - Teste de Armazenamento

é de apenas 15 GB, o que representa uma diferença de aproximadamente 91%. Mesmo considerando a quantidade de informação que foi descartada, é importante referir que, a informação processada e armazenada no sistema Elasticsearch, é suficiente para continuar a realizar todas as ações de monitorização e deteção de falhas.

Um dos principais problemas que afetavam a a solução anterior era o seu mau desempenho do painel de controlo. De modo a analisar de que forma é que a solução desenvolvida melhorou este desempenho, foram testados os tempos de carregamento de informação no painel de controlo antigo e no novo, usando para o efeito, um período de tempo semelhante.

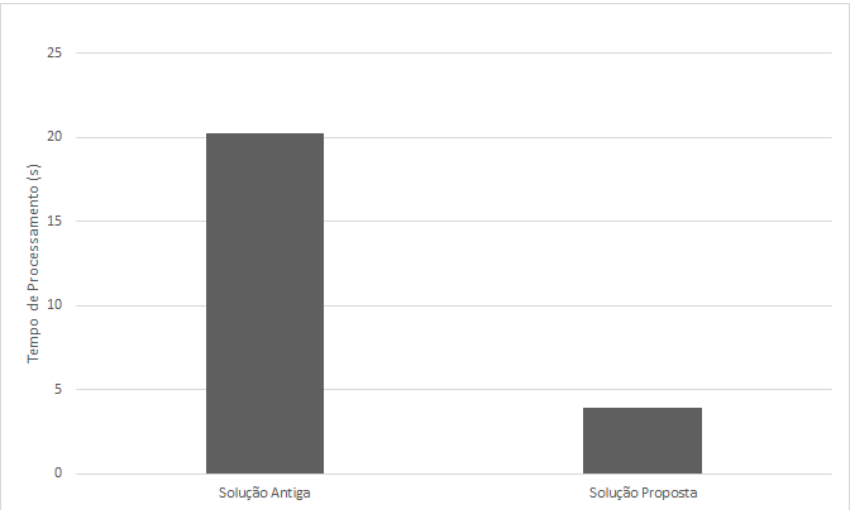


Figura 4.2: Resultados Finais - Teste de Velocidade

A figura 4.2 apresenta os resultados do teste descrito. Para o tempo de carregamento

do painel antigo foi registado um tempo de 20.27 segundos, enquanto que o novo processou uma quantidade semelhante de informação em apenas 3.93 segundos, o que representa uma diferença de aproximadamente 81%.

Um outro aspeto importante de medir é a usabilidade da solução. Uma vez que não é fácil encontrar valores ou métricas que meçam adequadamente os níveis de usabilidade de ambas as soluções, foi pedido a alguns colaboradores que experimentassem o novo painel de controlo e o comparassem com o antigo. De uma forma geral, o feedback recebido foi bastante positivo, sendo que afirmaram mesmo ser bastante mais fácil de navegar a informação dos registos, devido à interactividade do painel de controlo e rapidez de resposta a cada ação realizada.

4.2 Conclusões

Com todo o trabalho realizado neste projeto pensa-se ter conseguido implementar uma solução sólida e altamente configurável e extensível para a monitorização de vários tipos de sistemas. Apenas usando o código desenvolvido e a informação presente neste documento, pensa-se que seja suficiente para, não só perceber a forma como foram implementadas as várias partes do sistema e as razões por detrás das escolhas tomadas, mas também para perceber como desenvolver funcionalidade adicional ou configurar qualquer aspeto necessário para adaptar a solução a outros problemas.

Considerando os objetivos definidos inicialmente, apresentados em 1.4, o desenvolvimento do projeto foi bem sucedido. O primeiro desses objetivos era o de analisar ficheiros de registo de eventos provenientes de diferentes sistemas. Em 3.2 está descrito o trabalho realizado durante o período de desenvolvimento, onde foram analisados ficheiros de registos do sistema IIS e GENIO e definida uma estrutura de dados para cada um destes tipos de registos. Quanto a soluções de armazenamento, em 2.2 foi apresentado o estudo realizado a diferentes tipos destas soluções. Por demonstrar um melhor desempenho neste projeto, foi escolhido o sistema Elasticsearch que posteriormente foi instalado num ambiente virtual. Para processar os dados presentes nos ficheiros de registos e eventos, foi desenvolvido um conjunto de *scripts* em Python. Esta solução de processamento implementa todos os requisitos definidos inicialmente. Depois de extrair a informação dos ficheiros, essa informação é processada e inserida num objeto com a estrutura de dados definida anteriormente, sendo de seguida, enviada para o sistema Elasticsearch onde será armazenada. Adicionalmente, todo esse desenvolvimento foi baseado no paradigma de programação orientada a objetos, o que torna o código mais modular e fácil de configurar ou alterar. Finalmente, usando o sistema Kibana foi possível facilmente criar um painel de controlo com todas as funcionalidades necessárias para permitir a fácil leitura e manipulação de informação. Neste sistema é ainda possível filtrar a informação disponível de forma a facilitar o processo de descoberta de erros.

Relativamente a dificuldades ou limitações durante o projeto, a parte mais afetada foi possivelmente a parte de relacionar informação de vários tipos de sistemas. Durante o

período de desenvolvimento não foi possível obter registos do sistema ISS e da aplicação GENIO do mesmo período temporal, o que limitou bastante a análise de formas para relacionar a informação das duas fontes.

4.3 Trabalho Futuro

Usando esta solução apresentada como base, pensa-se haver um grande potencial para a realização de trabalho futuro. Um dos possíveis melhoramentos que poderão ser analisados ou implementados é a introdução de ferramentas de aprendizagem automática. Durante o documento foi referido várias vezes que este tipo de ferramentas foram consideradas durante o desenvolvimento do código. Deste modo, usando este tipo de ferramentas e, usando as funções de pré-processamento dos diferentes registos, poderão ser criados modelos de forma a definir a normalidade do desempenho do sistema. Usando estes modelos, posteriormente poderá ser possível classificar automaticamente os registos como normais ou anómalos.

Outro aspeto que pode ser desenvolvido é a parte de monitorização do sistema. Embora possa ser usada para a monitorização, o desenvolvimento da solução atual foi mais focado em deteção de falhas e identificação de problemas. Usando o painel de controlo implementado como base, poderão ser desenvolvidos outros tipos de visualizações que, juntamente com a funcionalidade de atualização automática do painel e ferramentas para inicialização periódica do script Python (como por exemplo *cron jobs* em sistemas Linux), poderão ser usadas para realizar uma monitorização do sistema de uma forma quase em tempo real.

Bibliografia

[1] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin e S. W. Schlosser. "Log-based Architectures for General-purpose Monitoring of Deployed Code". Em: *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*. ASID '06. San Jose, California: ACM, 2006, pp. 63–65. ISBN: 1-59593-576-2. DOI: 10.1145/1181309.1181319. URL: http://doi.acm.org/10.1145/1181309.1181319.

[2] S. Chen, P. B. Gibbons, M. Kozuch e T. C. Mowry. "Log-based architectures: using multicore to help software behave correctly". Em: *Operating Systems Review* 45.1 (2011), pp. 84–91.

[3] Elastic. *Elasticsearch Bulk API*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-bulk.html (acedido em 20/09/2019).

[4] Elastic. *Elasticsearch introduction*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html (acedido em 20/09/2019).

[5] Elastic. *Helpers — Elasticsearch 7.0.0 documentation*. URL: https://elasticsearch-py.readthedocs.io/en/master/helpers.html#bulk-helpers (acedido em 20/09/2019).

[6] D. L. Iverson. "Inductive system health monitoring". Em: *NASA Technical Reports Server*. 2004. URL: https://ntrs.nasa.gov/search.jsp?R=20040068062.

[7] Y. Li e S. Manoharan. "A performance comparison of SQL and NoSQL databases". Em: *IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing - Proceedings*. Ago. de 2013, pp. 15–19. DOI: 10.1109/PACRIM.2013.6625441.

[8] MySQL. *MySQL 5.7 Reference Manual - Alternative Storage Engines*. URL: https://dev.mysql.com/doc/refman/5.7/en/storage-engines.html (acedido em 20/09/2019).

[9] U. WENDEL. *Aggregation features, Elasticsearch vs. MySQL (vs. MongoDB)*. URL: http://blog.ulf-wendel.de/2016/aggregation-features-elasticsearch-vs-mysql-vs-mongodb/ (acedido em 20/09/2019).